PERSONA: A personalized model for code recommendation

Tam The Nguyen*, Tung Thanh Nguyen

Department of Computer Science and Software Engineering, Auburn University, Auburn, Alabama, United States of America

* tam@auburn.edu

Abstract

Code recommendation is an important feature of modern software development tools to improve the productivity of programmers. The current advanced techniques in code recommendation mostly focus on the crowd-based approach. The basic idea is to collect a large pool of available source code, extract the common code patterns, and utilize the patterns for recommendations. However, programmers are different in multiple aspects including coding preferences, styles, levels of experience, and knowledge about libraries and frameworks. These differences lead to various usages of code elements. When the code of multiple programmers is combined and mined, such differences are disappeared, which could limit the accuracy of the code recommendation tool for a specific programmer. In the paper, we develop a code recommendation technique that focuses on the personal coding patterns of programmers. We propose PERSONA, a personalized code recommendation model. It learns personalized code patterns for each programmer based on their coding history, while also combines with project-specific and common code patterns. PERSONA supports recommending code elements including variable names, class names, methods, and parameters. The empirical evaluation suggests that our recommendation tool based on PERSONA is highly effective. It recommends the next identifier with top-1 accuracy of 60-65% and outperforms the baseline approaches.

1 Introduction

In software development, programmers must interact with different types of information and perform many activities to build an application. They constantly need to figure out which variables, objects, or methods to use next. Additionally, the number of objects and methods to use in the current project or libraries are often huge, which makes programmers impossible to remember all the usages. To help programmers work more effectively, modern integrated development environments (IDEs) offer code recommendation features. These tools help developers to complete the names of classes, methods, fields, and keywords. Murphy et al. [1] performed a study that indicates programmers could use the code recommender up to several times per minute when they develop applications in Eclipse.
However, the default code recommendation plugins inside current IDEs offer fairly limited functionalities. Firstly, the current recommendation tools often provide the ranking candidates based on alphabetical order. Certain candidates have a higher probability to appear than others, but might not be included at the top of the ranked list. The recommendation could be time-consuming if the number of candidates is big, and the user needs to move down the rank list to find what he wants. Secondly, the built-in tools often lack the consideration of code context when making a recommendation. For example, let us assume a user created a new URL object in the previous line, he is likely to create a HttpURLConnection object by calling the openConnection method on the newly created URL object. Thus, a tool should recognize the existence of the URL object as the context when making recommendations.

To further improve the effectiveness and usefulness of current code recommendation tools, multiple methods have been proposed [2–6]. Most of the techniques are motivated by the crowd-based approach. The approach focuses on the common code patterns of objects and methods that are shared among multiple programmers. The idea is to build a large dataset by collecting a large pool of available source code. Next, common code patterns are extracted or inferred from the dataset. In the recommendation phase, the current code context is used to match with learned code patterns to infer the recommendations.

At the same time, each programmer has certain coding preferences and styles. For example, a programmer could prefer to use CSVReader object to read a file, while others prefer to use BufferedReader. These coding preferences are referred to as personal coding patterns of programmers. In the crowd-based approach, while common code patterns are combined and inferred, such coding preferences are blurred. This could limit the accuracy of the code recommendation tool for a specific programmer. To capture the personal coding patterns, a code recommendation tool should take into consideration the code history of written the programmer. For example, which classes, objects, or code patterns that the programmer often uses. Providing such recommendations could improve the effectiveness and enhance the user satisfaction of the tool. Our preliminary study [7] shows a recommendation model that incorporates personal code patterns provides improvements in suggesting variable declaration and initialization code. Therefore, it is desirable to combine both personal and common code patterns to improve current code recommendation models.

In this paper, we propose PERSONA, a novel code recommendation model that focuses on the personal coding patterns of programmers while also combines with project-specific and common code patterns. As a personalized model, PERSONA is built and updated for each programmer. It is composed of three sub-models: PerCR, a model that captures personal code patterns of a programmer; ProCR, a model that captures the project-level code patterns that the programmer is working on; and GenCR, a general model that capture code patterns shared between multiple projects. PERSONA incorporates code patterns learned from the three sub-models together and utilizes those patterns for recommending code elements including variable names, class names, methods, and parameters.

PERSONA utilizes the fuzzy set theory [8] to model correlation/association between code elements. It defines a fuzzy set of potential recommendation candidates toward code elements that appear in the current code context. Each candidate has a membership score, which determines a certain degree of membership in the fuzzy set. The membership score is calculated based on various factors such as the code history of the programmer, the project he is working on, or common code patterns. The candidate with a higher membership score will be ranked higher in the recommendation list. The details of our approach are presented in Section 4.

To build the proposed recommendation model, we extract personalized object usage instances from the code history of a programmer. We use such data to train a personalized code recommendation model PerCR for the programmer. The code history of other
programmers in the current project is also extracted to train a project-level recommendation model ProCR. We also train GenCR, a general model to capture common code patterns on a large code corpus. Finally, we incorporate the sub-models together to build PersonA. Once trained, given the current editing code in which the programmer wants to invoke code recommendation, our recommendation tool extracts its context features and utilizes PersonA to compute the recommendation rank list. The details of our recommendation system are presented in Section 5.

We have conducted several evaluation experiments to evaluate the usefulness and effectiveness of the personalized code recommendation approach. In the evaluation, PersonA is trained on a big dataset, containing 14,807 Java projects across multiple domains, amounting to over 350 million lines of code in over 2 million files. Next, the model is evaluated on 10 large Java projects with the number of commits in each project is ranging from 23,000 to over 400,000. The evaluation results show that PersonA could achieve high accuracy in code recommendation. For example, when evaluating PersonA on a programmer, our approach has top-1 accuracy of 66% and top-3 accuracy of 74%. Furthermore, our model also outperforms the baselines significantly in top-1 accuracy in these experiments. It outperforms the first baseline by an average of 12-15%, and generates a gap of 4-6% when compared to the second baseline. We also show that the recommendation accuracy of PersonA improves over time as more code of the programmer is used to train. By incorporating three sub-models together, the PersonA performs reasonably well even if the code history of the programmers is thin in the project. The details of our evaluation process are presented in Section 6.

The key contributions of our paper include:

- We proposed PersonA, a lightweight code recommendation model that focuses on the personal coding patterns of programmers. PersonA is built and updated for each programmer. To learn personal coding patterns, it utilizes fuzzy logic to model correlation/association between code elements in the code history written by the programmer. PersonA also incorporates project-specific and common code patterns efficiently to further improve the recommendation accuracy.

- We implemented a robust code recommendation system based on PersonA. The system includes a module to extract the usages of variables, methods, classes, parameters from the code history of a programmer, as well as from a large codebase. The system is designed to train PersonA efficiently. Furthermore, it also allows PersonA to be re-trained easily to update the coding preferences of programmers as more training data becomes available.

- We performed an extensive evaluation that shows the effectiveness of the approach in code recommendation. PersonA is trained on a dataset containing 14,807 Java projects, with over 350 million lines. We evaluated the model on 10 big Java projects with the number of commits in each project is ranging from 23,000 to over 400,000. The evaluation results show that PersonA could achieve high accuracy in code recommendation and outperforms the baselines significantly. We also showed that the model could be re-trained and improves the recommendation accuracy over time as more code of the programmer is available for training.

The rest of the paper is organized as follows. The related work is presented in Section 2. In Section 3, we present the motivation of our approach. Next, we describe our proposed model in Section 4. The description of our code recommendation system is presented in Section 5. We present our evaluation in Section 6. Section 7 presents our discussion. Finally, we conclude the paper in Section 8.
2 Related work

There are various code recommendation techniques have been proposed over years, including [2–5, 9–11]. Bruch et al. [2] proposed three example based code completion systems where examples are extracted automatically from the example code base. SLANG [4] uses statistical language models such as n-grams, and RNNs to model application programming interface (API) method usages and recommend the next API method call. Grapacc [5] is a graph-based, pattern-oriented, context-sensitive code completion approach that models API usage patterns as frequent graph-based models. DroidAssist [3, 9] models the usage of API objects and methods based on Hidden Markov Models and provides recommendation on method calls. Precise [10] builds a parameter usage database based on the existing code base and recommends API parameters. Graphite [11] is an active code completion architecture that allows library developers to introduce interactive and highly-specialized code generation interfaces directly into the editor. Most of the current code recommendation techniques focus on modeling common code patterns from a large code base, then utilize the patterns to make recommendations.

The statistical approach for capturing rules and patterns in source code has become a hot research topic in software engineering in recent years. Hassan et al. [12] indicated “natural” software analytics based on statistical modeling will become one of the most important aspects of software analytics. Hindle et al. [13] shows that source code is repetitive and predictable like natural language and they adopted an n-gram model on lexical tokens to suggest the next token. SLAMC [14] represents code by semantic tokens, i.e. annotations of data types, method/field signatures, etc. rather than lexical tokens. SLAMC combines n-gram modeling of consecutive semantic tokens, topic modeling of the whole code corpus, and bi-gram of related API functions. Tu et al. [15] exploited the localness of source code. White et al. [16] proposed deep learning approach modeling source code. Allamanis and Sutton [17] train an n-gram language model on a giga-token source code corpus. NATURALIZE [18] uses an n-gram language model to learns the style of a codebase and suggest natural identifier names and formatting conventions. Jacob et al. [19] uses an n-gram model to learn code templates. Hidden Markov Model has been used to infer the next token from user-provided abbreviations [20] and detect coded information islands, such as source code, stack traces, and patches, from free text [21]. Maddison et al. [22] proposed tree-based generative models for source code. Hsiao et al. [23] learns an n-gram language model on program dependence graph and uses the model for finding plagiarized code pairs. Nguyen et al. [24] introduced GraLan, a graph-based statistical language model that learns common API usage (sub)graphs from source code.

Pattern mining approaches represent usage patterns using various data structures such as sequences, sets, trees, and graphs. JADET [25] extracted a usage model as a set of partial order pairs of method calls. MAPO [6] mined frequent API call sequences and suggests associated code examples. Wang et al. [26] mines succinct and high-coverage API usage patterns from source code. Acharya et al. [27] proposed an approach to mine partial orders among APIs. Buse and Weimer [28] propose an automatic technique for mining synthesizing succinct and representative human-readable API examples. Other techniques include mining associate rules [29], item sets [2], subgraphs [30, 31], code idioms [32], etc.

Several approaches have been proposed to improve code recommendation/completion tasks. Robbes et al. [33] gathered information to improve code recommendation by replaying the entire change history of programs with the completion engine. In [34], the authors proposed and implemented new strategies for sorting, filtering, and grouping APIs in the code recommendation popup pane to improve accuracy. Hill and Rideout [35] proposed a method to match the code fragment under editing with small similar-structure code segments that often appear in large software projects. The authors of [36, 37] proposed methods to suggest
source code examples to developers based on API documentation. Holmes and Murphy [38] described an approach to recommend code examples based on heuristically matching with the structure of the code.

In general, personalized models have been studied extensively in the fields of recommender systems [39, 40] and collaborative filtering [41, 42]. For example, Hwang et al. [40] proposed a new recommender system, which employs a genetic algorithm to learn personal preferences of customers and provide tailored suggestions. In software engineering, several personalized approaches have been proposed. Jiang et al. [43] developed a separate prediction model for each developer to predict software defects. In [44], the author proposed a personalized defect prediction framework that gives instant feedback to the developer at change level, based on historical defect and change data. Wang et al. [45] proposed a context-aware personalized task recommendation approach to aid dynamic worker decision in selecting crowd-testing tasks.

Fuzzy-based approaches have been proposed to solve problems in software engineering, such as bug triaging problem [46, 47], automatic tagging [48], bug categorization [49]. However, they focus on modeling textual software artifacts.

3 Motivation

Let us start with an example that explains the challenges when using the current code recommendation methods, and motivates our approach. Fig 1 shows a code recommendation scenario in which the programmer writes code to read a file. In the first line, he creates an InputStream object from the filename. Next, he creates an InputStreamReader object from the InputStream. Let us assume that he invokes code recommendation at the first of line 4. A code recommendation method based on the crowd-based approach would recommend creating a BufferedReader (line 5). This is because using a BufferedReader to read a file from an InputStreamReader is a common code pattern that is often shared between programmers. The model learns the pattern from mining a code corpus.

The preference of the programmer in the example is different. He prefers to use CSVReader object to read file instead of BufferedReader. He has been using CSVReader throughout his application development. Thus, an ideal code recommendation tool should prioritize the personal code patterns and recommends CSVReader (line 6).

The example shows that programmers have preferences and styles when coding including naming variables, using certain classes, objects, and methods, or applying certain coding patterns. Thus, such personal preferences should be taken into consideration when providing code recommendations as it could improve the effectiveness and enhance user satisfaction of the recommendation tool.

```java
1. // Load file
2. InputStream is = testContext.getAssets().open(filename);
3. InputStreamReader inputStreamReader = new InputStreamReader(is);
4. //
5. // BufferedReader bf = new BufferedReader(inputStreamReader);
6. // CSVReader csvReader = new CSVReader(inputStreamReader);
```

Fig 1. A code recommendation scenario.

https://doi.org/10.1371/journal.pone.0259834.g001
4 Model

In PersoNa, code recommendation is modeled as a ranking problem: given the current editing code $E$ in which a programmer is asking recommendations for a missing code element, $\Phi$ is the set of all possible recommendation candidates, find a candidate $c \in \Phi$ with the highest possibility to be filled in the current missing location.

The key idea of PersoNa is to rank potential candidates $c$ toward a set of context features $F$ in $E$ by modeling the correlation/association of $c$ with each context feature in $F$. The set of feature $F$ includes object types, method calls, variable names, and parameters that occur in $E$. If a candidate $c$ has a higher correlation with features in $F$, $c$ is considered to have a higher possibility and will be rank higher in the list.

For example, in Fig 1, in which the programmer invokes code recommendation at beginning of line 4. The goal of PersoNa is to rank CSVReader as the declaration type with highest possibility. The set of context features $F$ includes object types $\{\text{InputStream, InputStreamReader}\}$, variables $\{\text{is, testContext, inputStreamReader}\}$, method calls $\{\text{getAssets, open, InputStreamReader.new}\}$, and parameters $\{\text{fileName}\}$.

To model correlation/association between candidates and context features, PersoNa utilizes the fuzzy set theory [8]. It defines a fuzzy set of potential candidates toward a context feature as follows.

**Definition 1 (Potential candidate)** For a specific context feature $f$, a fuzzy set $C_f$ with an associated membership function $\mu_f()$, represents the set of potential candidates toward $f$, i.e. candidates that are highly correlated with $f$.

Fuzzy set $C_f$ is determined via a membership function $\mu_f()$ with values in the range $[0, 1]$. For a candidate $c$, the membership score $\mu_f(c)$ determines the certainty degree of the membership of $c$ in $C_f$, i.e. how likely does $c$ belong to the fuzzy set $C_f$. $\mu_f(c)$ represents the degree of association between $c$ and $f$. $\mu_f(c)$ also determines the ranking of $c$ toward $f$. If $\mu_f(c) > \mu_f(c')$ then $c$ is considered higher correlated to $f$ to $c'$. The membership score is often computed as follows.

**Definition 2 (Membership score)** The membership score $\mu_f(c)$ is computed as the correlation between the set $D_f$ representing usages of the context feature $f$, and the set $D_c$ representing usages of the candidate $c$:

$$\mu_f(c) = \frac{|D_f \cap D_c|}{|D_f \cup D_c|} = \frac{n_{f,c}}{n_f + n_c - n_{f,c}} \quad (1)$$

where, $n_f$ is the number of usages of the context feature $f$, $n_c$ is the number of usages of the candidate $c$, and $n_{f,c}$ is the number of times that the candidate $c$ co-occurs with $f$. As the Eq 1, the value of $\mu_f(c)$ is between $[0, 1]$. If $\mu_f(c) = 1$, then $c$ always occurs on the code snippets that contain $f$, thus, given a code snippet contains $f$, it is very likely that $c$ co-occurs. If $\mu_f(c) = 0$, it means that $c$ never occurs on code snippets that contains $f$, thus, given a code snippet contains $f$, it is unlikely to recommend $c$. In general, the more frequently $c$ co-occurs with $f$, the higher value of $\mu_f(c)$.

Based on the fuzzy logic framework described above, we develop three different code recommendation models. Each model has its membership score function (Eq 1) and is learned from different datasets. Finally, we incorporate the three models together to build PersoNa. Let us describe each model as follows.

### 4.1 Personalized code recommendation model

As demonstrated in Section 3, programmers have different coding preferences, styles, experience levels, and knowledge about libraries and frameworks. For example, a programmer might
prefer using certain classes or methods than others; some programmers prefer short variable names for a `BufferedReader` object such as `b` or `bf`, while others use long names such as `bufferedReader`, etc. In other words, there are personal code patterns that appear in the code written by a programmer. Thus, a code recommendation model that utilizes those personal code patterns could improve the code recommendation performance significantly. Based on this observation, we design a personalized fuzzy-based code recommendation model (or PerCR for short).

Let us assume a programmer $d$ is working on a project $P$, $H_d$ is the code history written by the programmer in the current project. The membership score in PerCR is defined as:

$$
\mu_d(f, c, H_d) = \frac{n_{fc}(H_d)}{n_f(H_d) + n_c(H_d)}
$$

(2)

where $\mu_d(f, c, H_d)$ represents the membership score of candidate $c$ in the fuzzy set $C_f$ of the context feature $f$, $n_f(H_d)$ represents the usages of $f$ in $H_d$, $n_c(H_d)$ represents the usages of $c$ in $H_d$, and $n_{fc}(H_d)$ represents the usages in $H_d$ which $f$ and $c$ co-occur. Normally, $n_f(H_d)$ is defined as the number of occurrences of $f$ in $H_d$. However, in PerCR we also want to model the change in code patterns of programmers over time. For example, a programmer might start by using the `BufferedReader` object to read file but as he writes more code, he gradually changes his preference to using `CSVReader`. Thus, in PerCR, $n_f(H_d)$ is computed as follows:

$$
n_f(H_d) = \sum_{s \in H_d} e^{-\frac{1}{\Delta t}}
$$

where $\Delta t = t_c - t_0$ is the time decay, $t_0$ is the timestamp in which the project start, $t_c$ is the timestamp in which $f$ occurs in $H_d$. The idea behind the formula is that the occurrence of $f$ later in the project has more influence over the previous occurrences. Similarly, we have:

$$
n_c(H_d) = \sum_{s \in H_d} e^{-\frac{1}{\Delta t}} \quad \text{and} \quad n_{fc}(H_d) = \sum_{s \in H_d, c} e^{-\frac{1}{\Delta t}}
$$

4.2 Project-level code recommendation model

When multiple programmers work on the same project, they read, share, and reuse the code of each other. Thus, the code written by a programmer could be influenced by other programmers in the same project. For example, a programmer could create and use `MapUtil` class that contains several utility methods for `Map`. Other programmers in the same project also reuse the class. Thus, the code patterns related to the class could be shared between programmers in the project. We present ProCR, a fuzzy-based model that captures the project-level code patterns in the project that the programmer is working on.

Let us assume a programmer $d$ is working on a project $P$, and $P - H_d$ is the code history written by all other programmers (except $d$) in the project. ProCR is the project-level code recommendation model defined specifically for the programmer $d$. The membership score in ProCR is defined as:

$$
\mu_d(f, c, P - H_d) = \frac{n_{fc}(P - H_d)}{n_f(P - H_d) + n_c(P - H_d)}
$$

(3)

where $\mu_d(f, c, P - H_d)$ represents the membership score of candidate $c$ in the fuzzy set $C_f$ of the context feature $f$. Other terms in Eq 3 are defined similarly to corresponded terms in Eq 2. In other words, the project-level model ProCR is defined similarly to the personalized model.
4.3 General code recommendation model

In modern application development, programmers rely heavily on shared APIs to write code. For example, two different programmers could use the same API classes such as BufferedReader, File to read data from a file. The usage pattern of using those objects could be similar between the two programmers. Programmers might also share programming conventions of programming languages such as naming conventions. Thus, programmers do share common code patterns and we want to incorporate these patterns to our approach to improve the recommendation accuracy. We propose GenCR, a fuzzy-based model that captures such common code patterns shared between multiple projects. The membership score of GenCR is defined as follows:

\[ \mu(f, c, \mathcal{P}) = \frac{n_{f, c}(\mathcal{P})}{n_{f}(\mathcal{P}) + n_{c}(\mathcal{P}) - n_{f,c}(\mathcal{P})} \]  

where \( \mathcal{P} \) is the set contains the code of all projects in the dataset except the current project \( P \), \( n_{f}(\mathcal{P}) \) is the number of occurrences of \( f \) in \( \mathcal{P} \), \( n_{c}(\mathcal{P}) \) is the number of occurrences of \( c \) in \( \mathcal{P} \), and \( n_{f,c}(\mathcal{P}) \) is the number of times that the candidate \( c \) co-occurs with \( f \) in \( \mathcal{P} \).

4.4 Combining sub-models

Using each sub-model described above separately could yield a low-accuracy recommendation. For example, if a programmer just joined the project or the project just started, there is not much data to train PerCR and ProCR. Thus, these models could be fairly inaccurate, while GenCR could not recommend personal or project-level code patterns. To maximize the recommendation accuracy, we design Persona to incorporate the three sub-models together. It defines the membership score \( \mu_{d}(c) \) in Persona as follows:

\[ \mu_{d}(f, c) = a_1 \mu_{d}(f, c, H_d) + a_2 \mu_{d}(f, c, P - H_d) + a_3 \mu(f, c, \mathcal{P}) \]  

where \( a_1 + a_2 + a_3 = 1 \) are weighting coefficients. The value of \( a_i \) represents the contribution level of a sub-model towards Persona, the higher value of \( a_i \) the bigger contribution of the sub-model. If the model defines the membership score \( \mu_{d}(c) \) using Eq 5, we call the model PersonaSum.

As the sub-models are defined in separated datasets, the membership score of Persona could also be defined using the max function:

\[ \mu_{d}(f, c) = \max(\mu_{d}(f, c, H_d), \mu_{d}(f, c, P - H_d), \mu(f, c, \mathcal{P})) \]  

In Eq 6, the sub-model with the highest value of membership score will decide the value of \( \mu_{d}(f, c) \). If the model defines the membership score using Eq 6, we call the model PersonaMax. We experimented both approaches of calculating \( \mu_{d}(f, c) \) in Persona in our evaluation.

After defining the membership score function, we show how Persona calculates the rank list of candidates using the fuzzy set theory. Based on the definition of potential candidates toward a context feature \( f \) as a fuzzy set (Definition 1), Persona defines potential candidates toward a set of context features \( F \) using the union operation of fuzzy set theory as follows.

**Definition 3** Given a set of context features \( F \), a fuzzy set \( \mathcal{C}_{F} \), with an associated membership function \( \mu_{F}(\cdot) \), represents the set of potential candidates toward \( F \), i.e. the candidates that are highly correlated with context features of \( F \). \( \mathcal{C}_{F} \) is computed as the union of the fuzzy sets \( \mathcal{C}_{f} \) of
context features in $F$:

$$C_F = \bigcup_{F \in \mathcal{F}} C_f$$

Because $C_F$ is a fuzzy set, it has a membership function $\mu_F$. The union operation in fuzzy logic is defined via calculating $\mu_F$ from $\mu_{f_1} \ldots \mu_{f_k}$. There are several equations for fuzzy union operation, we use the following one:

**Definition 4** The membership score $\mu_F(c)$ is calculated as the combination of the membership scores $\mu_f(c)$ of its associated context feature $f$:

$$\mu_F(c) = 1 - \prod_{f \in F} (1 - \mu_f(c))$$

In Eq 8, $\mu_F(c)$ represents the correlation of candidate $c$ toward a set of context features $F$. As the equation, we see that the value of $\mu_F(c)$ is also between $[0, 1]$ and represents the likelihood in which the candidate $c$ belongs to the fuzzy set $C_F$, i.e. the set of potential candidates for the set of context features $F$. If there is any method $f$ in $F$ with $\mu_f(c) = 1$, then $\mu_F(c) = 1$, or PERSONA considers that $c$ is very likely to occur on the code contains $F$ as $c$ always occurs on code contains $f$ in $F$. In general, the more context features $f$ in $F$ with high $\mu_f(c)$ values, the higher $\mu_F(c)$ is, or $c$ is more likely to occur on the code contains $F$.

In the code recommendation phase, PERSONA ranks candidates based on the value of $\mu_F(c)$ and provides the rank list for the user. The higher value of $\mu_F(c)$, the higher ranking of the candidate $c$ in the list.

5 System implementation

5.1 Overview

In this section, we briefly discuss the points in the design and implementation of our recommendation system. Fig 2 shows an overview of the system. Overall, it consists of 3 main components. The code history extractor is the component for extracting personalized object usage instances from the code history of a programmer. The model learner uses the extracted data to train and incorporate the three sub-models in PERSONA. Finally, the code recommender utilizes the personalized model to make recommendations on the current editing code.

5.2 Code history extractor

Because our recommendation techniques are learned from the personalized code history of programmers, we have built a code history extractor module for extracting usages of variables, methods, classes, and parameters of a programmer from his code development history. Typically, whenever a programmer adds a new code or updates the current existing code, he will submit a commit to the version control system. Fig 3 shows an example of code changes in a commit of a programmer. In the example, the programmer switched from using a `HttpServletResponse` object to a `HttpURLConnection` object. In our approach, we extract personalized code patterns from code changes in commits. In particular, for each code change in a commit of the programmer, the code history extractor will analyze the post-commit version, and extract object usages in the new code that the programmer added.

To extract the usages of variables, methods, classes, and parameters, the extractor uses GROUM (Graph-based Object Usage Model) [30] to represent the object usages in the source code.
Fig 2. Overview of the system.

https://doi.org/10.1371/journal.pone.0259834.g002
code. GROUP is a graph that represents the object usages in source code. It has two kinds of nodes: **object nodes** and **action nodes**. An object node represents an object. It is labeled by the name of the object type (e.g. `HttpURLConnection`). An action node represents a method call. It is labeled the method qualified name (e.g. `URL.openConnection`). There are two kinds of edges representing control flow between action nodes and data flow between action nodes and object nodes.

In GROUP, each object created or involved during the execution is represented as an object node. We also treat primitive variables as object nodes. Action nodes represent any action that is performed on object nodes. Action nodes could be object instantiations, method calls, data field accesses of one object, or other operations. Object nodes are labeled by class names (object nodes represent primitive variables are labeled by types). Action nodes of types object instantiations, method calls, or data field accesses are labeled as “C.m” where C is its class name and m is the method (or field) name. Other action nodes that represent operations are labeled as the name of the operation.

The control edges of GROUP are used to represent the temporal orders between action nodes. A control edge from an action node A to action node B means that A is executed before B in the execution path. Because GROUP is defined for each execution path, thus, there is only one temporal order between action nodes, which is represented by a set of control edges between action nodes. The data edges indicate the data dependencies between data nodes and action nodes. A data edge from object node A to action node B means that A is a parameter of the action that B represents. A data edge from an action node B to data node A means the action node B returns the object node A.

Fig 4 illustrates the GROUP of the code that the programmer added in the commit. Rectangle nodes are action nodes, while object nodes are represented as round rectangle nodes. Solid arrows represent the control edge between action nodes and dashed arrows represent data edges. The algorithms are used to extract GROUP from source code could be found at [3, 9, 30].

There are several advantages of using GROUP to represent and extract the usages of variables, methods, classes, and parameters. Firstly, it removes redundant information in code such as keywords (`try`, `return`, ...), or symbols (`=`, `+`, ..), and only focuses on important information such as objects, method calls. Secondly, GROUP avoids the problem of duplicated counting when extracting the occurrences of code elements and co-occurrences between code elements. For example, in Fig 4, the variable name `conn` appears multiple times in the code.
Using GROUM, all the appearances are traced back to a single object node. Thus, the occurrence of `conn` is counted as one and the co-occurrence between `conn` and other code elements is also counted as one.

To produce the training data for PERSONA, the extractor travels through nodes in GROUM and counts the occurrences and co-occurrences between code elements. Note that, for each occurrence of a code element, we also store the timestamp in which the programmer added it to the project. The time information is important when training the personalized model. To train the sub-model GENCR, we also developed a code extractor to extract GROUM from source files in a code corpus.

5.3 Learning recommendation models

We train each sub-model in PERSONA separately. To train the personalized model PERCR, we need to calculate $n_c(H_d), n_f(H_d), n_i(H_d)$ in Eq 2. Calculating these values requires counting the occurrences of code elements and co-occurrences between code elements in the code history of the programmer. We explained the counting process in the previous section. Training the project-level model PROCR is similar to PERCR, the only difference is that PROCR is trained on the code history of other programmers in the current project. To train GENCR, we collect a code dataset contains multiple projects. Next, we obtain the source files from the projects and extract GROUM from source files. GENCR is trained by computing values $n_{c,(P)}, n_{f,(P)}, n_{i,(P)}$ described in Eq 4. Finally, we incorporate the three sub-models with either Eqs 5 or 6.
5.4 Recommending code

Let us go back to the scenario in Fig 1. The programmer writes code to read a file. In the first line, he creates an InputStream object from the filename. Next, he creates an InputStreamReader object from the InputStream. Let us assume, he invokes code recommendation at the first of line 4. Upon the request, our tool will analyze the current editing code, build a temporary Group, and extract the set of context features \( F \) includes object types \{InputStream, InputStreamReader\}, variables \{is, testContext, InputStreamReader\}, method calls \{getAssets, open, InputStreamReader.new\}, and parameters \{fileName\}.

In the next step, the tool will build a set of candidates for recommendation. It starts by analyzing which types of code elements are asked for the recommendation. In the example, the candidates should be a class or a variable. All classes and variables that are available in the current editing code will be added to the set of candidates. The tool then utilizes PERSONA to calculate the relevance score of each recommendation candidate towards the set of context features \( F \) using Eq 8. The set of candidates will be sorted by the relevance score. Finally, the recommendation tool returns the rank list of candidates with relevant scores for the programmer to consider. Note that, if the user requests recommendations for a new variable name, the tool will consider all the variable names has been used for the object before as the candidates. These names are stored in our model and might not be in the editing code.

6 Evaluation

We conducted several experiments to evaluate the effectiveness of our approach to learning and recommending code for programmers. All experiments are executed on a computer running Windows 10 with Intel Core i7 3.6Ghz CPU, 16GB RAM, and 1TB HDD storage. To conduct the evaluation, we collected a dataset consists of multiple Java projects that have source code repositories available on GitHub. The dataset that we used was carefully collected and studied by Allamanis et al. [17]. The corpus can be found and downloaded online [50]. It contains 14,807 projects across a wide variety of domains amounting to over 350 million lines of code in over 2 million files. The number of code tokens in the dataset exceeds 1.5 billion. Note that the dataset only contains the .java extension files, it does not contains revisions or commit changes. The characteristics of the corpus are shown in Table 1. We call this dataset A14K.

Because the A14K dataset only contains a snapshot of .java files, it is only used for training GENCER and the baselines. For evaluating PERSONA, it is required to have the code history of projects and programmers. Thus, we manually selected 10 projects in the dataset to evaluate our model. We selected such projects by first sorting the projects in the dataset by the number of commits. Next, we chose projects that have the highest number of commits while the vast majority of code is written in Java. We avoided selecting certain projects. First, we avoided projects that share duplicated code with a previously selected project. We also avoided projects that are developed in multiple programming languages. For each selected project, we checked out its source code repository to retrieve all the code and commit changes. Table 2 shows the list of selected projects along with the number of contributors and commits.

Table 1. Dataset characteristics.

| Number of projects | 14,807 |
|--------------------|--------|
| Number of files    | 2,130,264 |
| Number of lines of code | 352,312,696 |
| Number of code tokens | 1,501,614,836 |

https://doi.org/10.1371/journal.pone.0259834.t001
6.1 Settings and baselines

For each selected project, the set of commits is sorted in chronological order. Next, we grouped commits by programmers. When we perform an evaluation experiment for a programmer $d$, his commit set is divided into a training set $TR_d$ and testing set $TE_d$ chronologically. The training set $TR_d$ is used to train the sub-model $P_{ERCR}$. The code in the testing set $TE_d$ is used to evaluate later. In our experiments, the training set is selected as the first 70% of the initial set of commits in chronological order, while the remaining 30% of the commits are used as the testing set. The sub-model $P_{ROCR}$ is trained on the set of commits of other programmers. The commits that are used to train $P_{ERCR}$ appear before the first commit of the testing set $TE_d$. The sub-model $GENCR$ is trained on the initial dataset, which contains a snapshot of .java files (the current project is excluded). Finally, we combine the sub-models using both Eqs 5 and 6. In the first method, we set the weighting coefficients equally, $a_1 = a_2 = a_3 = \frac{1}{3}$. The second method uses the max function to combine sub-models. The two approaches are called $PERSONASUM$, and $PERSONAMAX$ correspondingly.

In our evaluation, to compare our model with the baselines, we chose the task of recommending the next identifier in a code sequence. The types of identifiers that we considered include variable and field names, type names such as class and interface names, method names, and parameters. Given a code sequence, a recommendation model is expected to recommend the most probable identifier. Alamalis et al. [17] shows that learning to predict code elements is difficult mainly because of the identifiers. Thus, we chose this task to better compare the effectiveness of recommendation models. This evaluation task has been used similarly in the evaluation of prior approaches [13–15].

Recommendation accuracy is measured as follows. Our evaluation tool predicts and evaluates all identifiers in every code sequence from the testing set. At a position $i$, it uses the recommendation model under evaluation to compute the top $k$ most likely identifiers $x_1, x_2, \ldots, x_k$ for that position based on the previous code tokens. If the actual identifier $s_i$ at position $i$ is among $k$ suggested results, we count this as a hit. The top-$k$ suggestion accuracy for a sequence is the ratio of the total hits over the sequence’s length. For example, if we have 70 hits on a code sequence of lengths 100 for a test file, accuracy is 70%. The top-$k$ accuracy is the ratio of the total hits over the total number of evaluated tokens.

To compare the effectiveness of $PERSONA$, we chose two baseline models: $n$-gram and recurrent neural network (RNN) for comparison due to the following reasons. First, both of them are popular statistical models to capture common patterns in a large dataset and are comparable with $PERSONA$. In addition, $n$-gram is widely used in recent research on code

| Name                | Description               | #Contributors | #Commits |
|---------------------|---------------------------|---------------|----------|
| liferay-portal      | Enterprise web platform   | 553           | 407,015  |
| intellij-comm       | IDE for Java              | 510           | 279,093  |
| osmand              | Android map app           | 688           | 62,493   |
| geogebra            | Multi-platform math app   | 24            | 52,750   |
| elasticsearch       | Search engine             | 1,415         | 52,628   |
| camel               | Middleware framework      | 609           | 45,158   |
| lucene-solr         | Enterprise-search platform| 181           | 33,654   |
| cdt                 | IDE for C++               | 266           | 34,603   |
| cassandra           | Database system           | 298           | 25,289   |
| hadoop              | Data processing framework | 278           | 23,877   |

Table 2. Projects used in the evaluation.

https://doi.org/10.1371/journal.pone.0259834.t002
recommendation [13–15], Raychev et al. [4] and White et al. [16] recently evaluated RNN and n-gram in code recommendation, and reported RNN as the better approach. Note that our model and the baselines use the same 14K dataset as the training set.

An n-gram model is a simple statistical model for modeling sequences. An n-gram model learns all possible conditional probabilities \( P(m_i|m_{i-n+1} \ldots m_{i-1}) \), where \( m_i \) is the current code token and \( m_{i-n+1} \ldots m_{i-1} \) is the sub-sequence of \( n-1 \) prior tokens. This is the probability that \( m_i \) occurs as the next code tokens of \( m_{i-n+1} \ldots m_{i-1} \). Using the chaining rule, we can use an n-gram model to compute the generating probability of any given sequence \( m_1 \ldots m_n \). To improve the effectiveness of n-gram model, Tu et al. [15] introduced CacheLM, a novel cache language model that consists of both an n-gram and an added “cache” component to exploit localness. The cache is the set of code tokens that appear in the same project as the test file. We re-implement this method for comparison. We used the same settings with the original model, i.e., 3-gram with "5K tokens" cache size. The model is trained on the A14K dataset with the cache is extracted from the project under test.

A Recurrent neural network (RNN) is a class of neural networks for learning sequences. A single-layer RNN can be trained with a collection of code token sequences and can compute the probability of the next code token for any given sequence. In other words, the RNN can compute all conditional probabilities \( P(m_i|m_{i-1} \ldots m_{i-1}) \) for any given sequence \( m_1 \ldots m_n \). To do that, it maintains a context vector (hidden state) \( c_i \) represents current context of sub-sequence up to \( m_1 \ldots m_{i-1} \). A function \( f \) is learned from data to compute the context vector at position \( i \), \( c_i = f(m_i, c_{i-1}) \) given the current token \( m_i \) and previous context \( c_{i-1} \) while another function \( g \) is learned to compute the probability of the next token \( m_{i+1} \), \( P(m_{i+1}|m_1 \ldots m_i) = g(c_i) \) given the current context \( c_i \). To improve the modeling performance, we could stack multiple layers of RNNs on top of each other to create a Deep RNN. Each hidden state is continuously passed to both the next time step of the current layer and the current time step of the next layer. The model could still be further improved by using a special type of the hidden layer called Long Short-Term Memory (LSTM) cell to tackle the problem of unstable gradients and handle long sequences. A Deep RNN model with too many hidden layers is quite computationally expensive. Thus, in our experiment, we implemented a model with a stack of 5 hidden states, each hidden state is an LSTM cell with 200 hidden units. We call this model DRNN200-5. We implemented DRNN200-5 using Keras Sequential APIs, TensorFlow 2, and running a Google Colab Pro machine. Note that, the code sequences are used to train both CacheLM and DRNN200-5 are extracted using the GROUM model as we described in Section 5.

### 6.2 Recommendation accuracy

In this section, we show the recommendation accuracy of our proposed models and the baselines on top-contributed programmers over 10 selected projects. For each project, we select top-5 contributed programmers by the number of commits. We train and test our models and the baselines for each programmer. Due to a lack of space, we only report top-1 accuracy. Tables 3 and 4 show the top-1 commendation accuracy of top-contributed programmers in the projects IntelliJ-comm and osmand. To report the result in all the selected projects, we compute the average top-1 accuracy of programmers in each project. Table 5 shows the average results in all 10 projects.

From the tables, we can see several interesting results. Overall, the personalized model PerCR outperforms the project-level recommendation model ProCR, and the general model GenCR. It generates a 2-3% gap over ProCR and 12-15% gap over GenCR. When incorporating sub-models together, the recommendation accuracy increases significantly with top-1 accuracy approaching 60-65%. The combining method using weighting coefficients
Table 3. Recommendation accuracy of top-contributed programmers in IntelliJ-comm.

| Name                  | #Commits | PERSONACR | PROCR | GENCR | PERSONASUM | PERSONAMAX | CacheLM | DRNN200-5 |
|-----------------------|----------|-----------|-------|-------|------------|------------|---------|-----------|
| Dmitry Jemerov        | 15,962   | 58.5%     | 55.7% | 41.7% | 64.3%      | 63.6%      | 47.0%   | 57.5%     |
| Peter                 | 14,615   | 55.9%     | 53.2% | 39.2% | 66.0%      | 62.9%      | 48.0%   | 58.3%     |
| Alexey Kudratsev      | 13,934   | 59.3%     | 55.3% | 40.1% | 64.9%      | 63.0%      | 47.1%   | 57.9%     |
| Anna Kozlova          | 12,589   | 57.5%     | 55.8% | 39.8% | 66.0%      | 61.4%      | 46.3%   | 58.4%     |
| Vladimir Krivosheev   | 11,784   | 55.7%     | 57.0% | 40.8% | 66.6%      | 62.5%      | 47.0%   | 58.8%     |

Table 4. Recommendation accuracy of top-contributed programmers in osmand.

| Name            | #Commits | PERSONACR | PROCR | GENCR | PERSONASUM | PERSONAMAX | CacheLM | DRNN200-5 |
|-----------------|----------|-----------|-------|-------|------------|------------|---------|-----------|
| Victor Shcherb  | 6,730    | 53.8%     | 49.6% | 39.1% | 63.1%      | 61.4%      | 48.2%   | 55.4%     |
| Weblate         | 4,029    | 50.4%     | 48.8% | 38.3% | 61.8%      | 58.4%      | 47.0%   | 55.3%     |
| Sonora          | 3,586    | 51.9%     | 47.8% | 37.9% | 60.5%      | 60.9%      | 47.2%   | 54.4%     |
| Franco          | 2,536    | 52.8%     | 49.2% | 34.3% | 59.4%      | 60.8%      | 45.3%   | 53.2%     |
| Jan Madsen      | 2,207    | 50.1%     | 46.8% | 38.5% | 61.7%      | 62.6%      | 46.4%   | 56.6%     |

(PERSONASUM) yields a slightly higher result compared to using the max function (PERSONAMAX) but the difference is insignificant.

About the baselines, the top-1 accuracy of DRNN200-5 is significantly higher than CacheLM with a gap of around 10%. This shows that DRNN200-5 is a much better approach for modeling sequences. The top-1 accuracy of DRNN200-5 is also better when compared with each sub-model. However, when combining the sub-models, PERSONASUM still has higher top-1 accuracy than DRNN200-5 by an average of 4-6%. Overall, the result shows that by combining three simple sub-models that capture personal, project-specific, and common code patterns together, PERSONA still outperforms the baselines which mostly focus on the common code patterns.

6.3 Recommendation accuracy over time

In this section, we evaluate the recommendation accuracy of PERSONA and sub-models over time. We design the experiment as follows. For a programmer d, we divide his set of commits into equal time intervals. A time interval of $t_i$ contains all the commits of the programmer during that time. Depending on the code history of programmers, we could divide the commits into months, quarters, or years. At time interval $t_i$, we will use all the commits of the programmer before $t_i$ to train PERCR, and all the commit of other programmers before $t_i$ to train

Table 5. Recommendation accuracy of top-contributed programmers in selected projects.

| Project           | PERCR | PROCR | GENCR | PERSONASUM | PERSONAMAX | CacheLM | DRNN200-5 |
|-------------------|-------|-------|-------|------------|------------|---------|-----------|
| liferay-portal    | 53.7% | 51.5% | 37.5% | 65.3%      | 63.4%      | 46.9%   | 57.9%     |
| intellij-comm     | 57.4% | 55.4% | 40.3% | 65.6%      | 62.7%      | 47.1%   | 58.7%     |
| osmand            | 51.8% | 48.0% | 37.6% | 61.3%      | 60.8%      | 46.8%   | 54.3%     |
| georgebra         | 52.6% | 48.5% | 35.5% | 61.7%      | 59.8%      | 46.7%   | 54.4%     |
| elasticsearch     | 46.8% | 43.8% | 36.9% | 58.4%      | 56.4%      | 42.0%   | 54.6%     |
| camel             | 49.5% | 45.6% | 38.8% | 58.1%      | 56.8%      | 43.3%   | 53.2%     |
| lucene-solr       | 50.2% | 49.1% | 40.9% | 60.5%      | 58.3%      | 42.5%   | 54.9%     |
| cdt               | 48.3% | 44.9% | 37.9% | 57.2%      | 55.5%      | 44.1%   | 51.8%     |
| cassandra         | 49.5% | 45.5% | 37.6% | 56.1%      | 55.2%      | 43.5%   | 52.4%     |
| hadoop            | 50.5% | 46.7% | 40.9% | 60.6%      | 58.5%      | 45.7%   | 54.7%     |

https://doi.org/10.1371/journal.pone.0259834.t003

https://doi.org/10.1371/journal.pone.0259834.t004

https://doi.org/10.1371/journal.pone.0259834.t005
**ProCR.** In other words, PERSONA is trained on all the commits of the project before $t_i$. All the commits in $t_i$ will be used for testing. With this experiment, we want to replicate the real-world accuracy of code recommendation models.

We choose intellij-comm as the subject system. The project has 279,093 commits with 510 contributors spanning from 2005 to now. From the project, we select three programmers with different types of contributions for evaluation. Due to the lack of presentation space, we will use quarters as the time interval, and we only show the recommendation accuracy for the first 20 intervals. Figs 5–7 show the top-1 recommendation accuracy over time for three programmers in the project. For better visualization, we only show the top-1 accuracy of GENCR, PROCR, and PERSONA in the figures. The top-1 accuracy of GENCR and the baselines is similar to values reported in Table 3.

From the figures, we can see that the recommendation accuracy of PROCR and PROCR increase over time as these models have more training data. This leads to an increase in the accuracy of PERSONA. Another interesting observation is that the amount of training data affects PROCR and PROCR significantly. The first programmer (Fig 5) is the main contributor to the project from the start. As he committed a lot of code, his personalized model outperforms the project-level model. The second programmer (Fig 6) joined the project when it already contains most of the code. Thus, his project-level recommendation model outperforms the personalized model in the beginning. Finally, the third programmer (Fig 7) has a limited
contribution at the beginning of the project. His personalized model has low accuracy at the beginning due to a lack of training data. Overall, the experiment shows that the recommendation accuracy of \textsc{Persona} improves over time as more training data is available.

6.4 Accuracy on lower-contributed programmers

We have studied the recommendation accuracy of our models on top-contributed programmers. In this section, we study how our models perform when recommending code for lower-contributed programmers. Of course, we do not want to select programmers that committed too little code, as we want to ensure that we have enough training and testing data for the personalized model. Thus, we select the programmers to evaluate as follows. First, for each project, we filter out all programmers with less than 20 commits. Next, we sort programmers by the number of commits and find the median of the list. We select five programmers that have the number of commits right above the median for the study. We train and test our models and the baselines for each programmer. To report the result in all the projects, we compute the average top-1 accuracy of the five programmers in each project.

Table 6 shows the average results in all 10 projects. We could see that the top-1 accuracy of the personalized model \textsc{PerCR} is low due to the lack of training data. The project-level model \textsc{ProCR} still performs reasonably well when compared to other models. This could be explained that the lower-contributed programmers often join the project later when the project has been
developed extensively, and they might reuse the project-specific code. Thus, the accuracy of the recommendation model is maintained. On average, PERSONA still achieves the highest top-1 accuracy when compared to the baselines. For example, PERSONA\text{SUM} has higher top-1 accuracy than DRNN200-5 in 8 out of 10 selected projects with a gap of around 2-5\%, while DRNN200-5 just slightly outperforms in the remaining 2 projects. Overall, by incorporating three sub-

![Fig 7. Top-1 accuracy over time of “Alexey Kudravtsev”](https://doi.org/10.1371/journal.pone.0259834.g007)

Table 6. Recommendation accuracy of lower-contributed programmers in selected projects.

| Project      | PerCR  | ProCR | GenCR  | PERSONA\text{SUM} | PERSONAMAX | CACHELM | DRNN200-5 |
|--------------|--------|-------|--------|--------------------|------------|---------|-----------|
| liferay-portal | 26.5\% | 48.7\% | 37.5\% | 58.6\%             | 56.8\%     | 46.2\% | 51.3\%    |
| intellij-comm | 24.2\% | 51.4\% | 39.8\% | 57.3\%             | 57.1\%     | 49.8\% | 54.4\%    |
| osmand       | 22.7\% | 45.3\% | 35.7\% | 57.0\%             | 52.8\%     | 47.2\% | 53.4\%    |
| geogebra     | 20.7\% | 44.8\% | 35.5\% | 56.3\%             | 54.1\%     | 46.0\% | 53.7\%    |
| elasticsearch| 15.3\% | 41.4\% | 36.9\% | 47.2\%             | 49.7\%     | 41.1\% | 47.8\%    |
| camel        | 20.1\% | 43.6\% | 38.8\% | 52.4\%             | 47.9\%     | 42.9\% | 49.6\%    |
| lucene-solr  | 17.5\% | 46.5\% | 40.9\% | 53.9\%             | 52.5\%     | 42.0\% | 51.3\%    |
| cdt          | 21.7\% | 42.0\% | 37.9\% | 48.3\%             | 48.7\%     | 43.5\% | 49.2\%    |
| cassandra    | 15.6\% | 42.9\% | 37.6\% | 51.0\%             | 49.6\%     | 42.6\% | 49.3\%    |
| hadoop       | 16.5\% | 43.1\% | 40.9\% | 55.6\%             | 52.4\%     | 44.9\% | 51.4\%    |

https://doi.org/10.1371/journal.pone.0259834.t006
models together, the \textsc{persona} performs reasonably well even if the programmers have low contribution in the project or did not join the project for a long time.

### 6.5 Ablation study

In this section, we perform an ablation study to understand the contribution of the sub-models to the recommendation performance of \textsc{persona}. In particular, we focus on the model \textsc{persona}_{\text{SUM}}. Similar to the first experiment, we measure the recommendation accuracy of \textsc{persona}_{\text{SUM}} with different configurations on top-contributed programmers over 10 selected projects. For each project, we select top-5 contributed programmers by the number of commits, then we compute the average top-1 accuracy of programmers in each project. As described in Section 4, in \textsc{persona}_{\text{SUM}}, we combine sub-models using Eq 5 where $\alpha_1 + \alpha_2 + \alpha_3 = 1$ are weighting coefficients. The values of $\alpha_1$, $\alpha_2$, $\alpha_3$ represent the contribution level of \textsc{per}, \textsc{pro}, \textsc{gen} correspondingly. Removing a sub-model from the system equals to setting $\alpha_i = 0$. For example, if we remove \textsc{pro} from the system, $\alpha_2$ is set to 0, which means $\alpha_1 = \alpha_3 = \frac{1}{2}$. The model is called \textsc{per}+\textsc{gen}. Similarly, if we remove \textsc{pro} and \textsc{gen} from the system, $\alpha_2$ and $\alpha_3$ are set to 0, which means $\alpha_2 = 1$. The model become \textsc{per}.

Table 7 shows the average top-1 recommendation accuracy when removing one or two sub-models from the system. Note that, as we use the same settings as the previous experiment, the results for \textsc{per}, \textsc{pro}, \textsc{gen}, \textsc{persona}_{\text{SUM}} are the same as in Table 5. We have several interesting observations. Firstly, if we remove two sub-models from the system, the sub-model \textsc{per} outperforms the project-level recommendation model \textsc{pro}, and the general model \textsc{gen}. Secondly, if we remove a sub-model from the system, \textsc{per}+\textsc{gen} has the highest top-1 accuracy. \textsc{per}+\textsc{gen} also has a significantly higher top-1 accuracy when compared to then each sub-model, especially, \textsc{gen}. This result shows that although \textsc{gen} has low top-1 accuracy, combining this sub-model with others could improve the recommendation significantly. Finally, we could see that \textsc{per}+\textsc{pro} does not have much improvement when compared to each sub-model.

### 6.6 Weighting coefficients

In this section, we study how the values of weighting coefficients affect the recommendation result of the model. As a personalized model, \textsc{persona} is built and updated for each programmer. We selected the top-1 programmer by the number of commits in the \textsc{intellij-comm} project to study the weighting coefficients. Let us assume we choose the weighting coefficient $\alpha_i$ to study. Note that the weighting coefficients have a constraint $\alpha_1 + \alpha_2 + \alpha_3 = 1$. For each

| Project      | \textsc{per} | \textsc{pro} | \textsc{gen} | \textsc{per}+\textsc{pro} | \textsc{pro}+\textsc{gen} | \textsc{per}+\textsc{gen} | \textsc{persona}_{\text{SUM}} |
|--------------|--------------|--------------|--------------|----------------------------|----------------------------|----------------------------|----------------------------|
| liferay-portal | 53.7%        | 51.5%        | 37.5%        | 56.6%                      | 55.8%                      | 61.1%                      | 65.3%                      |
| intellij-comm | 57.4%        | 55.4%        | 40.3%        | 59.1%                      | 58.3%                      | 60.7%                      | 65.6%                      |
| osmand        | 51.8%        | 48.0%        | 37.6%        | 54.3%                      | 50.2%                      | 55.9%                      | 61.3%                      |
| geogebra      | 52.6%        | 48.5%        | 35.5%        | 53.9%                      | 52.1%                      | 56.8%                      | 61.7%                      |
| elasticsearch | 46.8%        | 43.8%        | 36.9%        | 48.9%                      | 47.3%                      | 53.8%                      | 58.4%                      |
| camel         | 49.5%        | 45.6%        | 38.8%        | 51.1%                      | 48.5%                      | 53.4%                      | 58.1%                      |
| lucene-solr   | 50.2%        | 49.1%        | 40.9%        | 52.6%                      | 51.9%                      | 57.4%                      | 60.5%                      |
| cdt           | 48.3%        | 44.9%        | 37.9%        | 50.6%                      | 48.6%                      | 54.0%                      | 57.2%                      |
| cassandra     | 49.5%        | 45.5%        | 37.6%        | 49.9%                      | 49.2%                      | 53.8%                      | 56.1%                      |
| hadoop        | 50.5%        | 46.7%        | 40.9%        | 51.5%                      | 50.8%                      | 56.1%                      | 60.6%                      |

https://doi.org/10.1371/journal.pone.0259834.t007
value of $\alpha_1$, we set $\alpha_2 = \alpha_3 = \frac{1-\alpha_1}{2}$. Next, we let $\alpha_1$ takes different values from 0 to 1, and increase by 0.1. Then we evaluate the top-1 accuracy of PERSONASUM based at each value of $\alpha_1$. Fig 8 shows how top-1 accuracy changes when $\alpha_1$ changes. A similar process is repeated for the remaining weighting coefficients. Figs 9 and 10 show the result for $\alpha_2$ and $\alpha_3$.

From the figures, we have several observations. Firstly, if a weighting coefficient has a high value (closer to 1), the top-1 accuracy tends to decrease. In such a case, the top-1 accuracy of PERSONASUM is dominant by a sub-model. Secondly, the top-1 accuracy result is most sensitive with the value of $\alpha_3$. When $\alpha_3$ is high, the result decreases significantly as the weighting of GENCR increases in PERSONASUM. Additionally, we could see that the result is often high if all the weighting coefficients are in the range [0.3, 0.5]. These observations are valuable in selecting the values of weighting coefficients to improve the recommendation accuracy of PERSONA.

7 Discussion

In this section, we discuss several aspects of PERSONA in more detail. From the machine learning perspective, PERSONA is a simple ensemble approach with three sub-models: PerCR, a model that captures personal code patterns of a programmer; ProCR, a model that captures the project-level code patterns that the programmer is working on; and GenCR, a general model that capture code patterns shared between multiple projects. PERSONA learns and
recommends like n-gram and RNN but more flexibly. For example, fuzzy membership functions are not probability distribution functions, thus, PERSONA does not need to normalize $\sum \mu_f(c) = 1$.

We could consider PERSONA is a fuzzy logic system specially designed for the software engineering domain. It represents code patterns as fuzzy logic rules. It uses fuzzy set theory to model and apply those rules, and uses fuzzy union operations to combine the rules. In the traditional fuzzy logic system, variables are often continuous such as Temperature, Density or linguistic like LOW, VERY LOW. The membership functions are often manually defined by domain experts with functions such as triangular or trapezoidal. In PERSONA, variables are discrete, i.e. class, method, etc., and the membership functions are estimated automatically.

In our evaluation, we re-implemented (CACHELM) as the baseline method. Although we tried to replicate the same settings as the previous research [4, 15], the recommendation results of the baseline models in our evaluation are different when compared to the original research. The dissimilarity could be explained due to the differences in several factors including the dataset, cross-validation, recommendation tasks, etc. Similarly, in our implementation of DRnn200-5, we used different configurations with the previous studies [4, 16] so the result is not comparable.
The evaluation suggests that PERSONA outperforms the baseline models such as DRNN200-5 which reaffirms our earlier assumption. As a crowd-based approach, DRNN200-5 infers and recommends common code patterns from a large code corpus while ignoring the difference in coding preferences of programmers. When such differences are blurred, the performance of the recommendation tool for a specific programmer is hurt. PERSONA achieves high accuracy because it takes into consideration the personal coding preferences of programmers while also captures the project-specific and common code patterns. In our future work, we plan to incorporate personal coding patterns with the model such as DRNN200-5 to further improve the recommendation accuracy.

In PERSONA, we combine the sub-models using both Eqs 5 and 6. In the first method, we set the weighting coefficients equally, $\alpha_1 = \alpha_2 = \alpha_3 = \frac{1}{3}$. These weighting coefficients determine the contribution of a sub-model to PERSONA. We performed a study on how values of weighting coefficients affect the recommendation result of the model, which reveals several insights. Different combinations of weighting coefficients could be experienced to optimize the performance of the model. In our future work, we plan to develop a method to estimate such optimal coefficients.

![Fig 10. Top-1 accuracy when $\alpha_3$ changes.](https://doi.org/10.1371/journal.pone.0259834.g010)
8 Conclusion

To help programmers work more productive, modern IDEs often include code recommendation features. There are multiple techniques have been proposed to further improve the effectiveness of current recommendation tools. Most of the methods focus on modeling and extracting common code patterns that often appear in a large pool of available source code. However, each programmer has certain coding preferences and styles. These preferences are personal and might differ between programmers. Such personal preferences should be taken into consideration when providing code recommendations as it could improve the effectiveness and enhance user satisfaction. We propose PERSONA, a novel personalized code recommendation model while also combines with project-specific and common code patterns. As a personalized model, PERSONA is built and updated for each programmer. It utilizes the fuzzy set theory to model correlation/association between code elements. The empirical evaluation suggests that our recommendation tool based on PERSONA is highly effective. It outperforms the baseline models in the task of recommending the next identifier in a code sequence.

Acknowledgments

We thank the Department of Computer Science and Software Engineering at Auburn University for supporting us for this project.

Author Contributions

Data curation: Tam The Nguyen.
Methodology: Tam The Nguyen.
Project administration: Tam The Nguyen.
Software: Tam The Nguyen.
Supervision: Tung Thanh Nguyen.
Validation: Tam The Nguyen.
Visualization: Tam The Nguyen.
Writing – original draft: Tam The Nguyen.
Writing – review & editing: Tam The Nguyen.

References

1. Murphy GC, Kersten M, Findlater L. How are Java software developers using the Eclipse IDE? IEEE Software. 2006; 23(4):76–83. https://doi.org/10.1109/MS.2006.105
2. Bruch M, Monperrus M, Mezini M. Learning from Examples to Improve Code Completion Systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE’09. New York, NY, USA: Association for Computing Machinery, 2009. p. 213–222. Available from: https://doi.org/10.1145/1595696.1595728.
3. Nguyen TT, Pham HV, Vu PM, Nguyen TT. Learning API Usages from Bytecode: A Statistical Approach. In: Proceedings of the 38th International Conference on Software Engineering. ICSE’16. New York, NY, USA: Association for Computing Machinery; 2016. p. 416–427. Available from: https://doi.org/10.1145/2884781.2884873.
4. Raychev V, Vechev M, Yahav E. Code Completion with Statistical Language Models. SIGPLAN Not. 2014; 49(6):419–428. https://doi.org/10.1145/2666356.2594321
5. Nguyen AT, Nguyen TT, Nguyen HA, Tamrawi A, Nguyen HV, Al-Kofahi J, et al. Graph-based pattern-oriented, context-sensitive source code completion. In: 2012 34th International Conference on Software Engineering (ICSE); 2012. p. 69–79.
6. Zhong H, Xie T, Zhang L, Pei J, Mei H. MAPO: Mining and Recommending API Usage Patterns. In: Proceedings of the 23rd European Conference on ECOOP 2009—Object-Oriented Programming. Genoa, Berlin, Heidelberg: Springer-Verlag; 2009. p. 318–343. Available from: https://doi.org/10.1007/978-3-642-03013-0_15.

7. Nguyen T, Vu P, Nguyen T. Personalized Code Recommendation. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME); 2019. p. 313–317.

8. Klir GJ, Yuan B. Fuzzy Sets and Fuzzy Logic: Theory and Applications. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.; 1995.

9. Nguyen TT, Pham HV, Vu PM, Nguyen TT. Recommending API Usages for Mobile Apps with Hidden Markov Model. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. ASE’15. IEEE Press; 2015. p. 795–800. Available from: https://doi.org/10.1109/ASE.2015.109.

10. Zhang C, Yang J, Zhang Y, Fan J, Zhang X, Zhao J, et al. Automatic Parameter Recommendation for Practical API Usage. In: Proceedings of the 34th International Conference on Software Engineering. ICSE’12. IEEE Press; 2012. p. 826–836.

11. Omar C, Yoon YS, LaToza TD, Myers BA. Active code completion. In: 2012 34th International Conference on Software Engineering (ICSE); 2012. p. 859–869.

12. Hassan AE, Hindle A, Runeson P, Shepperd M, Devanbu PT, Kim S. Roundtable: What’s Next in Software Analytics. IEEE Software. 2013; 30(4):53–56. https://doi.org/10.1109/MS.2013.85

13. Hindle A, Barr ET, Su Z, Gabel M, Devanbu P. On the Naturalness of Software. In: Proceedings of the 34th International Conference on Software Engineering. ICE’12. Piscataway, NJ, USA: IEEE Press; 2012. p. 837–847. Available from: http://dl.acm.org/citation.cfm?id=2337223.2337322.

14. Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN. A Statistical Semantic Language Model for Source Code. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2013. New York, NY, USA: ACM; 2013. p. 532–542. Available from: http://doi.acm.org/10.1145/2491411.2491458.

15. Tu Z, Su Z, Devanbu P. On the Localness of Software. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014. New York, NY, USA: ACM; 2014. p. 269–280. Available from: http://doi.acm.org/10.1145/2635868.2635875.

16. White M, Vendome C, Linares-Vásquez M, Poshyvanyk D. Toward Deep Learning Software Repositories. In: Proceedings of the 12th Working Conference on Mining Software Repositories. MSR’15. Piscataway, NJ, USA: IEEE Press; 2015. p. 334–345. Available from: http://dl.acm.org/citation.cfm?id=2820518.2820559.

17. Allamanis M, Sutton C. Mining source code repositories at massive scale using language modeling. In: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on; 2013. p. 207–216.

18. Allamanis M, Barr ET, Bird C, Sutton C. Learning Natural Coding Conventions. In: Proceedings of the ACM SigSoft Symposium on Foundations of Software Engineering. ACM—Association for Computing Machinery; 2014. Available from: http://research.microsoft.com/apps/pubs/default.aspx?id=228792.

19. Jacob F, Tairas R. Code Template Inference Using Language Models. In: Proceedings of the 48th Annual Southeast Regional Conference. ACM SE’10. New York, NY, USA: ACM; 2010. p. 104:1–104:6. Available from: http://dl.acm.org/citation.cfm?id=1900008.1900143.

20. Han S, Wallace DR, Miller RC. Code Completion from Abbreviated Input. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. ASE’09. Washington, DC, USA: IEEE Computer Society; 2009. p. 332–343. Available from: http://dx.doi.org/10.1109/ASE.2009.64.

21. Cerulo L, Ceccarelli M, Di Penta M, Canfora G. A Hidden Markov Model to detect coded information islands in free text. In: Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on; 2013. p. 157–166.

22. Maddison CJ, Tarlow D. Structured Generative Models of Natural Source Code. In: The 31st International Conference on Machine Learning (ICML); 2014. Available from: http://research.microsoft.com/apps/pubs/default.aspx?id=208387.

23. Hsiao CH, Cafarella M, Narayanasamy S. Using Web Corpus Statistics for Program Analysis. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages Applications. OOPSLA’14. New York, NY, USA: ACM; 2014. p. 49–65. Available from: http://doi.acm.org/10.1145/2660193.2660226.

24. Nguyen AT, Nguyen TN. Graph-Based Statistical Language Model for Code. In: Proceedings of the 37th International Conference on Software Engineering—Volume 1. ICSE’15. IEEE Press; 2015. p. 858–868.
A personalized code recommendation model

25. Wasylkowsk i A, Zeller A, Lindig C. Detecting Object Usage Anomalies. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ESEC-FSE'07. New York, NY, USA: ACM; 2007. p. 35–44. Available from: http://doi.acm.org/10.1145/1287624.1287632.

26. Wang J, Dang Y, Zhang H, Chen K, Xie T, Zhang D. Mining succinct and high-coverage API usage patterns from source code. In: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on; 2013. p. 319–328.

27. Acharya M, Xie T, Pei J, Xu J. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ESEC-FSE'07. New York, NY, USA: ACM; 2007. p. 25–34. Available from: http://doi.acm.org/10.1145/1287624.1287630.

28. Buse RPL, Weimer W. Synthesizing API Usage Examples. In: Proceedings of the 34th International Conference on Software Engineering. ICSE'12. Piscataway, NJ, USA: IEEE Press; 2012. p. 782–792. Available from: http://dl.acm.org/citation.cfm?id=2337223.2337316.

29. Lo D, Maoz S. Mining Scenario-Based Triggers and Effects. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. ASE'08. Washington, DC, USA: IEEE Computer Society; 2008. p. 109–118. Available from: http://dx.doi.org/10.1109/ASE.2008.21.

30. Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi JM, Nguyen TN. Graph-Based Mining of Multiple Object Usage Patterns. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ESEC/FSE'09. New York, NY, USA: Association for Computing Machinery; 2009. p. 383–392. Available from: https://doi.org/10.1145/1595696.1595767.

31. Chang RY, Podgurski A, Yang J. Discovering Neglected Conditions in Software by Mining Dependence Graphs. Software Engineering, IEEE Transactions on. 2008; 34(5):579–596. https://doi.org/10.1109/TSE.2008.24.

32. Allamanis M, Sutton C. Mining Idioms from Source Code. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014. New York, NY, USA: ACM; 2014. p. 472–483. Available from: http://doi.acm.org/10.1145/2635868.2635901.

33. Robbes R, Lanza M. How Program History Can Improve Code Completion. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. ASE'08. USA: IEEE Computer Society; 2008. p. 317–326. Available from: https://doi.org/10.1109/ASE.2008.42.

34. Hou D, Fletcher DM. An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM); 2011. p. 233–242.

35. Hill R, Rideout J. Automatic method completion. In: Proceedings. 19th International Conference on Automated Software Engineering. 2004.; 2004. p. 228–235.

36. McMillan C, Pushyavany D, Grechanik M. Recommending Source Code Examples via API Call Usages and Documentation. In: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering. RSSE'10. New York, NY, USA: Association for Computing Machinery; 2010. p. 21–25. Available from: https://doi.org/10.1145/1808920.1808925.

37. Subramanian S, Iozenmetsva L, Holmes R. Live API Documentation. In: Proceedings of the 36th International Conference on Software Engineering. ICSE 2014. New York, NY, USA: Association for Computing Machinery; 2014. p. 643–652. Available from: https://doi.org/10.1145/2568225.2568313.

38. Holmes R, Murphy GC. Using structural context to recommend source code examples. In: Proceedings. 27th International Conference on Software Engineering. 2005. ICSE 2005.; 2005. p. 117–125.

39. Chu W, Park ST. Personalized Recommendation on Dynamic Content Using Predictive Bilinear Models. In: Proceedings of the 18th International Conference on World Wide Web. WWW'09. New York, NY, USA: Association for Computing Machinery; 2009. p. 691–700. Available from: https://doi.org/10.1145/1526709.1526802.

40. Hwang CS, Su YC, Tseng KC. Using Genetic Algorithms for Personalized Recommendation. In: Pan JS, Chen SM, Nguyen NT, editors. Computational Collective Intelligence. Technologies and Applications. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010. p. 104–112.

41. Song X, Tseng BL, Lin CY, Sun MT. Personalized Recommendation Driven by Information Flow. In: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. SIGIR'06. New York, NY, USA: Association for Computing Machinery; 2006. p. 509–516. Available from: https://doi.org/10.1145/1148170.1148258.

42. Shen S, Hu B, Chen W, Yang Q. Personalized Click Model through Collaborative Filtering. In: Proceedings of the Fifth ACM International Conference on Web Search and Data Mining. WSDM'12. New York,
43. Jiang T, Tan L, Kim S. Personalized defect prediction. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2013. p. 279–289.

44. Eken B. Assessing Personalized Software Defect Predictors. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings. ICSE’18. New York, NY, USA: Association for Computing Machinery; 2018. p. 488–491. Available from: https://doi.org/10.1145/3183440.3183449.

45. Wang J, Yang Y, Wang S, Chen C, Wang D, Wang Q. Context-aware Personalized Crowtesting Task Recommendation. IEEE Transactions on Software Engineering, 2021; p. 1–1. https://doi.org/10.1109/TSE.2021.3081171

46. Tamrawi A, Nguyen TT, Al-Kofahi JM, Nguyen TN. Fuzzy set-based automatic bug triaging: NIER track. In: 2011 33rd International Conference on Software Engineering (ICSE); 2011. p. 884–887.

47. Tamrawi A, Nguyen TT, Al-Kofahi JM, Nguyen TN. Fuzzy Set and Cache-based Approach for Bug Triaging. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ESEC/FSE’11. New York, NY, USA: ACM; 2011. p. 365–375. Available from: http://doi.acm.org/10.1145/2025113.2025163.

48. Al-Kofahi JM, Tamrawi A, Nguyen TT, Nguyen HA, Nguyen TN. Fuzzy set approach for automatic tagging in evolving software. In: Software Maintenance (ICSM), 2010 IEEE International Conference on; 2010. p. 1–10.

49. Chawla I, Singh SK. An Automated Approach for Bug Categorization Using Fuzzy Logic. In: Proceedings of the 8th India Software Engineering Conference. ISEC’15. New York, NY, USA: ACM; 2015. p. 90–99. Available from: http://doi.acm.org/10.1145/2723742.2723751.

50. Allamanis M, Sutton C. GitHub Java Corpus.; Available from: http://groups.inf.ed.ac.uk/cup/javaGithub/.