# Assessing Developer Expertise from the Statistical Distribution of Programming Syntax Patterns

Arghavan Moradi Dakhel
Polytechnique Montreal
Montreal, Quebec, Canada
arghavan.moradi-dakhel@polymtl.ca

Michel C. Desmarais
Polytechnique Montreal
Montreal, Quebec, Canada
michel.desmarais@polymtl.ca

Foutse Khomh
Polytechnique Montreal
Montreal, Quebec, Canada
foutse.khomh@polymtl.ca

## ABSTRACT

Accurate assessment of developer expertise is crucial for the assignment of an individual to perform a task or, more generally, to be involved in a project that requires an adequate level of knowledge. Potential programmers can come from a large pool. Therefore, automatic means to provide such assessment of expertise from written programs would be highly valuable in such context.

Previous works towards this goal have generally used heuristics such as *Line 10 Rule* or linguistic information in source files such as comments or identifiers to represent the knowledge of developers and evaluate their expertise. In this paper, we focus on syntactic patterns mastery as an evidence of knowledge in programming and propose a theoretical definition of programming knowledge based on the distribution of Syntax Patterns (SPs) in source code, namely Zipf's law. We first validate the model and its scalability over synthetic data of *"Expert"* and *"Novice"* programmers. This provides a ground truth and allows us to explore the space of validity of the model. Then, we assess the performance of the model over real data from programmers. The results show that our proposed approach outperforms the recent state of the art approaches for the task of classifying programming experts.

## CCS CONCEPTS

• **General and reference** → *Evaluation.*

## KEYWORDS

knowledge assessment, syntax pattern, Zipf law, software maintenance, version control system

**ACM Reference Format:**
Arghavan Moradi Dakhel, Michel C. Desmarais, and Foutse Khomh. 2021. Assessing Developer Expertise from the Statistical Distribution of Programming Syntax Patterns. In *Evaluation and Assessment in Software Engineering (EASE 2021), June 21–23, 2021, Trondheim, Norway.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3463274.3463343

## 1 INTRODUCTION

Identifying the expertise of developers is important when searching for an individual to hire, to solve a bug, or to contribute in a software project. The cost of poor recruiting decisions is estimated to be as high as ten times the annual salary of an employee [7]. While platforms such as GitHub offer a valuable information source about an individual's skills [23, 29], a survey study shows that 66% of non-technical recruiters have difficulty judging applicants based on their GitHub activities [37]. They struggle with technical information contained in these platforms; e.g., the manner or content of the applicants' participation in different projects. Thus, developing a model to automatically assess developers' knowledge in programming and identify experts is an important goal.

Representing a developer's knowledge from their written code is challenging. Previous works have relied on heuristics such as the *Line 10 Rule* and the number of commits in different source files [3, 25, 27], or the number of different API calls [31, 34] to represent the knowledge of developers. Another group of studies analyze textual and linguistic information within code, like comments or function names, to represent the domain knowledge of a developer [24, 36].

However, many of these aforementioned heuristics provide indirect evidence of knowledge and can prove unreliable or biased indicators of knowledge. In a preliminary study, the first author of this paper manually inspected 378 sample commits[1] of the Pandas repository[2] on GitHub and observed that in 102 (27%) *diff* files, despite the high number of code lines, some changes have no impact on the semantics of the program. For example, developers deleted additional space or they corrected the spelling of a variable in different lines. Therefore, in addition to the quantity of changes, such as the number of commits, the nature of the changes is also very important to capture the knowledge of a developer.

In this paper, we focus on mastery of programming language syntax patterns as a proxy of programming expertise. We identify expert developers by assessing their proficiency at using programming syntactic patterns. We assume that a part of developer's knowledge is defined by the subset of programming constructs a developer masters, such as syntactic patterns and lexical expressions in her/his artifacts.

The idea is also motivated by a theory in linguistics, named Zipf's law [44]. Zipf's law explains the distribution between words in a corpus. Baixeries et al [5] observed that the communication skills of an individual is reflected by the Zipf distribution of vocabularies in her/his conversation. Their assumption is consistent with finding different vocabulary size and distributions in the speech of various individuals. The model we propose relies on the Zipf distribution of syntactic patterns in artifacts produced by a developer to assess developer's mastery in programming syntax patterns and distinguish experts from novices. Also, we consider the number of

*EASE 2021, June 21–23, 2021, Trondheim, Norway*
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9053-8/21/06…$15.00
https://doi.org/10.1145/3463274.3463343

[1]https://www.surveysystem.com/sscalc.htm
[2]https://github.com/pandas-dev/pandas

distinct syntax patterns in a developer's artifacts as the vocabulary size in Zipf distribution and build a baseline by representing the expertise of a developer with this vocabulary size. Although syntactic patterns are superficial evidence of knowledge, the evaluation of our proposed model over real data shows that it achieves better performance in classifying experts from novices than different baselines and state of the art techniques. This model is therefore an important first step towards our community's long-term goal of developing an efficient technique for the automatic assessment of developers' expertise.

This paper makes the following contributions:

- We propose a model to assess developers' mastery in the programming syntactic patterns through parameter estimation of Zipf's law.
- We classify expert developers from novices based on their mastery in using syntax patterns and achieve better performance than recent state of the arts.

The remainder of this paper is organized as follows. Section 2 presents related works. Section 3 provides a brief background on Zipf distribution. Section 4 describes the whole process of our proposed approach. Section 5 explains the data, details our method to evaluate the proposed approach, shows experimental results, and answers research questions. Section 6 discusses different factors that constitute threats to the validity of our experiment. Section 7 concludes the paper and discusses avenues for future work.

## 2 RELATED WORK

Previous works to assess expertise in programming can be divided into two general groups: *"Internal expertise"* with aim of finding expert within a software project and *"Overall expertise"* focusing on assessing expertise of developers using their contributions in different software projects.

The majority of models to assess expertise from both categories are based on a set of features such as the number of commits, number of votes, or number of changes in a file path. A widely used heuristic is the *Line 10 Rule* [25, 27], inspired by version control systems that store the name of the author in line 10 of the commit log. All methods motivated by *Line 10 Rule* heuristic states that if a developer changed a file in the past, she/he should be one of the candidates to solve the tasks related to this file [3, 40]. Several studies find experts in Question Answering (Q&A) or Crowd Sourcing platforms. They use features such as the number of questions and answers, number of votes for different answers or the number of tasks performed by an individual to estimate her/his expertise [9, 13]. Other studies go further and collect different words in linguistic information of the codes such as words in comments, commits messages, the content of ReadMe files, or identifiers in the code to define the domain expertise of its authors. Then, they calculate the similarity between these words and the words in the description of bug reports or tasks to identify an expert to solve a bug or perform a task [20, 24, 36]. In Q&A platforms, the textual description or the tags of questions and answers are collected to represent the knowledge of members [4, 33]. Another group of studies collect API calls to determine the domain expertise of developers [15, 31, 34, 39].

To validate models, studies which focus on assessing *Internal Expertise* use bug resolution information and information about tasks performed by developers (from the revision history of projects) to assess the effectiveness of their models [1, 42]. However, this type of information is specific to a single project. One of the researches in the *Overall Expertise* category [28] performs a self-evaluation survey study to generate ground truth. But, they observed overestimation/underestimation in the survey results. Then, one of the authors manually searched a group of developers on Linkedin to verify their expertise. Another group of studies asked external experts such as group of students in the related field or the project managers to evaluate the result of their model [21, 39]. CVExplorer [20] validated their outcomes by recommending candidates for job positions in two companies.

Heuristic approaches that collect quantitative features such as number of commits, number of votes, number of API calls, or textual descriptions such as comments or identifiers, are sometimes effective indicators of the knowledge of developers, but they are not strongly grounded in theory and may be subject to biases. For example, Verdi et al. [41] represent that even a highly voted code snippet may have low quality and contain vulnerabilities. In this paper, we introduce a more theoretically grounded approach to assess the expertise of developers in syntactic mastery and compare its performance with state of the arts from the literature.

## 3 BACKGROUND

This section introduces a brief background on Zipf's law which is pivotal to the proposed approach.

Zipf's law is a rank-frequency distribution well known in linguistics [44]. George Kingsley Zipf discovered that the rank of words times their frequencies in a corpus is proportional to a constant [45]. Equation (1) shows this relation where $Pr(r_w; \alpha, n)$ is the probability of a word $w$ and $r_w$ is its rank, $n$ is the number of distinct words and $\alpha$ is an exponent close to 1 [43]. It is possible for a probability distribution to go with $\alpha < 1$, if $n$ is bounded by an above limit [10]. $H_n$ is an Harmonic series to normalize the equation.

$$Pr(r_w; \alpha, n) = \frac{1}{r_w^{\alpha} * H_{n,\alpha}} \tag{1}$$

If we take a logarithm on both sides of (1), then we have (2) which is a linear function. It converts a power law function into a line in a log-log space with intercept $\log H_{n,\alpha}$ and slope $\alpha$.

$$\log Pr(r_w) = -\alpha \log r_w - \log H_{n,\alpha} \tag{2}$$

Zipf's law is known to model the word distribution of natural languages, English, German, Chinese and Russian for example [32]. More recent studies show that Zipf's law also applies to programming languages. Zhang et al. [43] found that the distribution of lexical tokens in Java, C++ and C source codes follows a Zipf distribution. Other studies compared the Zipf distribution in human language and programming language [8, 35] and they find that the range of values for the $\alpha$ parameter in Zipf's law is different between human and programming languages.

In addition, there are studies showing a relation between expertise and Zipf law. There is a study that estimate the vocabulary growth of second-language learners with Zipf's law [16]. For example, an individual who is a beginner in speaking English may

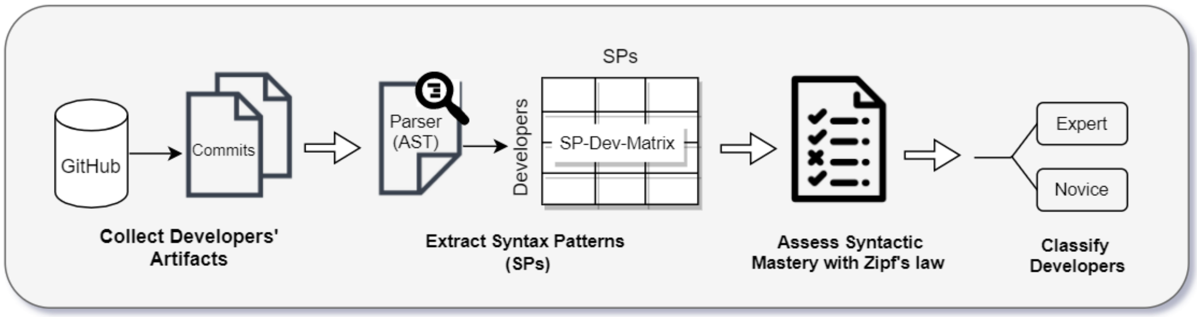

GitHub
Commits
Collect Developers' Artifacts
Parser (AST)
Developers
SPs
SP-Dev-Matrix
Extract Syntax Patterns (SPs)
Assess Syntactic Mastery with Zipf's law
Expert
Novice
Classify Developers

**Figure 1: The process of the proposed approach in classifying developers based on syntactic mastery**

express *"I am angry because I cannot fix this problem"* while she/he is not aware of more advanced sentences such as *"I find this problem frustrating!"*. Another study compared the speech of children and adults in terms of the Zipf distribution and showed that there is a dependency between the parameters of Zipf's law and the communication skills of individuals [5]. In this paper we explore a similar relationship between the parameters of Zipf's law and the distribution of syntax patterns in developers' code artifacts and use it to assess their syntactic mastery.

## 4 APPROACH

Fig. 1 presents an overview of our approach on how to represent the knowledge of developers with syntactical patterns and assess syntactic mastery with Zipf's law. Each step is described across the following sections.

### 4.1 Collect Developers' Artifacts

A program is a product of many contributors. Version control systems such as GitHub allow us to retrieve the artifacts of programmers in different repositories over time. The combination of commits written by a developer on different projects on GitHub are good representative of her/his experience. We collect the commits of developers on source files (i.e., all files with ".py" extension) across their contributions in different projects to define their knowledge state.

### 4.2 Extract Syntax Patterns

In the theory of knowledge space, an individual's knowledge state is defined as a subset of a knowledge domain [17]. In this study, this domain is defined as a set of syntactical patterns.

Every commit has a number of elements such as API calls, identifiers, or lexical expressions and programming language keywords. The way that developers use programming language keywords and expressions builds different Syntax Patterns (SPs) in their artifacts. Developers may use different methods and consequently different syntactic patterns to solve a problem based on their knowledge state. For example, Fig. 2 shows two different solutions to solve the same problem in the Python language. In this study, we collect programming keywords and lexical expressions in commits as knowledge items of its author. We use the Abstract Syntax Tree (AST) to extract and collect the SPs.

```
def plus(x):
    return x+2

output=[]
for i in range(1,5):
    output.insert(i,plus(i))

print(output)
```

**(a)**

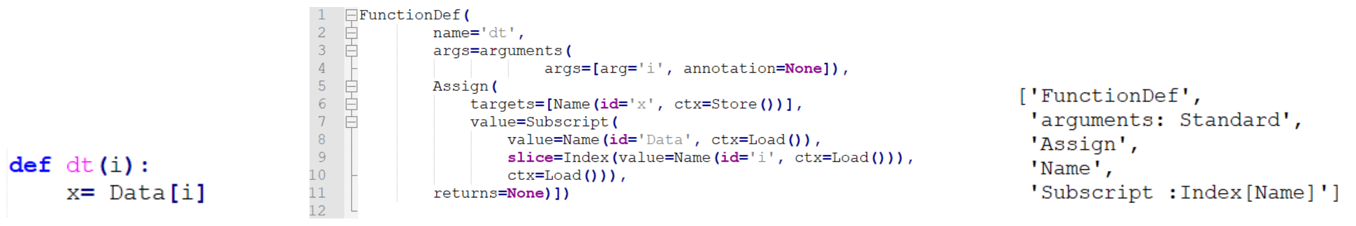
```
print(list(map(lambda x : x+2, range(1,5))))
```

**(b)**

**Figure 2: (a) and (b) are two different methods of adding a variable (between 1 to 5) with 2, in Python**

***Abstract Syntax Tree (AST):*** Abstract Syntax Tree is an abstract version of the parsing tree [11]. The AST is larger than the natural sentence of the code in vocabulary size. There are new nodes in the AST that represents the relationship between the abstract components of the programming code [30]. For example, Fig. 3b shows the AST dump of the Python code in Fig. 3a. We collect nodes in AST as Syntactical patterns. Fig. 3c shows the SPs that we collect from the sample code in Fig. 3a. In this figure, for example,{*'Subscript: Index[Name]'*}, means accessing a subscript with indexing type of a variable. We omit the lowest layer in each path of AST which are for example variable names or function names (textual information). While the lowest layer may represent project specific knowledge which are not common in different repositories, we chose to limit the domain to more generic knowledge items because our goal is to assess their proficiency at using SPs.

There is a file, named *diff*, attached to each commit in Git that contains all lines that a developer touched in a source file. The exact change made by a developer is not specified in the *diff* file. To address this limitation, we compare the AST of a source file before and after applying a commit [34] and we collect the difference (added/removed) nodes among two trees as knowledge items of its author. With this method, we ensure that developers actually practiced the SPs that we are collecting as their knowledge. At the end of this step, we build a matrix of developers and the frequency of different SPs in their commits.



```
def dt(i):
    x= Data[i]
```

(a) Sample python code

```
FunctionDef(
        name='dt',
        args=arguments(
                args=[arg='i', annotation=None]),
        Assign(
            targets=[Name(id='x', ctx=Store())],
            value=Subscript(
                value=Name(id='Data', ctx=Load()),
                slice=Index(value=Name(id='i', ctx=Load())),
                ctx=Load())),
        returns=None)])
```

(b) AST dump

```
['FunctionDef',
 'arguments: Standard',
 'Assign',
 'Name',
 'Subscript :Index[Name]']
```

(c) Syntax Patterns

**Figure 3: AST and a list of Syntax Patterns for a sample Python code. (a) shows a sample code in python, (b) is the AST of sample code in (a), and (c) represents the Syntax Patterns (SPs) collected from AST in (b)**

## 4.3 Assess Syntactic Mastery with Zipf's law

In this section, we describe the relation between syntactic mastery and Zipf's law exponents. Fig. 4 shows the probability distribution of SPs in commits of two developers with different knowledge states in Python. However both charts show a Zipf distribution with a few high probability SPs and a fat tail of low probability SPs, but they are not mirroring the same behavior. It represents that the number of SPs and the distribution between them is different among two sample developers. The probability of the top rank SP, $r = 1$, for the expert sample is around 0.07 while it is 0.05 for the novice sample. Expert sample shows a longer tail or more advanced patterns while the frequency of middle SPs are higher for novice sample. It is worth mentioning that these are not necessarily the same patterns.

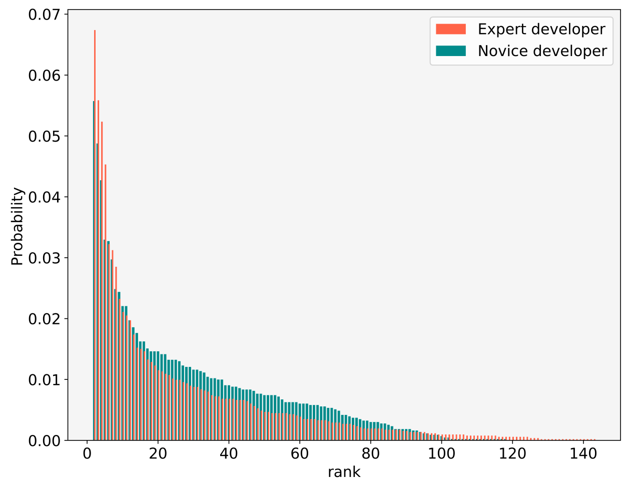


**Figure 4: Probability distribution of SPs of two developers with different knowledge states**

*What is the relation between Zipf's law and Mastery of SPs?*

Let us explain with an example, what we mean by the relationship between Zipf's law and programming syntax mastery. Suppose that $dev_1$ and $dev_2$ are two developers with the same level of proficiency at programming syntax and that they have used similar SPs with the same frequency in their commits so far. After fitting the distribution of SPs in their commits with Zipf's law, the $\alpha = 1.072$ for both developers.

Suppose that we ask both developers to write code to solve a new problem, for example, a function to add a number in range 1 to 5 with 2. One of the developers, $dev_1$, chooses the approach in Fig. 2a. $dev_1$ shows three new SPs, {*'Add','List','insert'*} and repeats most of her/his current knowledge items. However, $dev_2$ chooses the method in Fig. 2b. $dev_2$ shows more new knowledge items. The new items disclosed by $dev_2$ are {*'map','Lambda','Add','List'*}. We fit the new distribution of SPs for each developer with Zipf's law and we find that the parameter $\alpha$ for $dev_1$ is decreased to $\alpha = 1.02$ and for $dev_2$ is increased to $\alpha = 1.19$. The bars in Fig. 5 shows the real distribution of SPs and the lines are the fitted Zipf's law for these distributions. It is evident that different values are assigned to exponent $\alpha$ to fit different distributions with Zipf's law. Since developers choose different SPs to write code based on their knowledge, our assumption is that there is a dependency among developers' mastery of SPs and exponent $\alpha$.

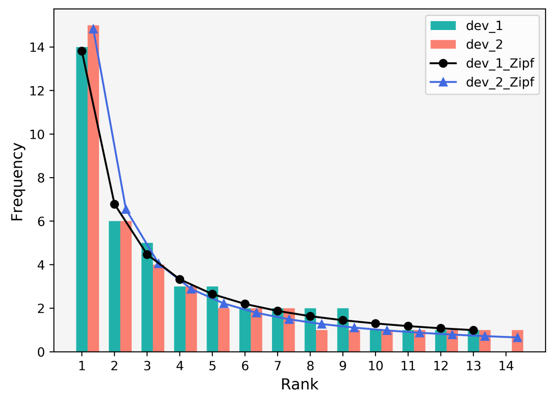


**Figure 5: The Zipf distribution between SPs of two sample developers**

## 4.4 Classify Expert

We focus on two categories of developers: "Expert" and "Novice". "Expert" developers demonstrate a strong mastery in programming, using SPs and solving programming bugs. On contrast, "Novice" developers have few number of contributions and use basic SPs more frequently. We distinguish experts from novices based on their level of mastery of SPs. We fit the distribution of SPs in developers' commits with Zipf's law and estimate the $\alpha$ per each developer.

Then, we use the value of this exponent to classify experts from novices. We call the proposed method as *"zipf_α"*.

We setup two types of classification tasks: one where we do not know the proportion of experts/novices and another where the proportion is determined from a labeled training set. They respectively correspond to two assessment methods: *"Clustering"* and *"Classifiers"*.

*Clustering.* In the clustering group, we use Jenks Natural Breaks Clustering (JNBC) [22, 26] for methods with single-dimensional data such as our proposed method, *"zipf_α"*, that uses only one feature, estimated value of $\alpha$, to illustrate the syntactic mastery.

*Classifier.* In the classifier group, since this is a binary classification problem, we simply find from the training set the optimum threshold [6] for exponent $\alpha$ to classify experts from novices.

## 5 EVALUATION OF THE PROPOSED APPROACH

In this section we describe our dataset. Then, we explain how to fit a distribution with Zipf's law and estimate the goodness of fit. Subsequently, we illustrate a list of methods that we compare the performance of the proposed model in identifying experts with them. Finally, we report our results and answer the following two research questions:

*RQ1: How does expertise impact the distribution between SPs in developers' commits?*

*RQ2: How well can Zipf's law assess developers' mastery in programming SPs and discriminate experts from novices?*

### 5.1 Dataset

To illustrate the properties of our model we use both synthetic and real datasets. Given that the real data is not labeled and manual inspection of huge number of developers is costly and time consuming, we generate a labeled synthetic dataset of 1200 developers based on real sample developers. We use this data to calibrate the difference between distribution of SPs in the code of high number of developers. Also we use this to find the optimum number of developers that we need to manually inspect from real data.

*Real dataset.* We collect commits of developers on source files in GitHub to define their knowledge state. We focus on Python programming language because it is a common programming language in different fields, nowadays. A lot of people without programming or software background write code in Python due to the importance of Artificial Intelligence (AI) and Machine Learning (ML) in all majors domains. We use pydriller [38] to collect GitHub data. We use the following rules to collect data of developers with different levels of expertise. Table 1 shows a summary of collected data.

(1) Start with top 10 Python projects on GitHub based on their number of stars like Python [3] and Keras [4]
(2) Collect the top 10 and the last 10 developers in the list of contributors based on the number of commits in each repository
(3) Link each author name to her/his profile on GitHub based on their aliases or usernames
(4) For each repository in her/his profile: Collect last year commits (for those developers who didn't meet the optimum sample size of the Zipf's law, we collect more commits over a period of multiple years until reach the optimum size)
(5) Collect the top 50 and the last 50 python developers in GitHub based on the number of commits in projects with python programming language (to have more diversity between developers and projects)
(6) Repeat step 4 for each of the 100 developers from step 5

[3]https://github.com/Python/cPython
[4]https://github.com/keras-team/keras

**Table 1: A summary of real dataset collected from github**

| # of Commits | # of Projects | # of Developers |
|---|---|---|
| 54676 | 441 | 300 |

We obtain a total of 54676 commits. For each of these 54676 commits, we compare the AST of the source file before and after applying the commits and we discover 197 unique SPs.

*Generate synthetic data.* We generate a labeled dataset in two categories of *"Expert"* and non-expert or *"Novice"*. To do so, first, we manually select one expert developer from our dataset of real developers. Then, we calculate the probability of all SPs in the commits of this sample developer. We use these probabilities as a weight to generate SPs in commits of synthetic *Expert* developers. For those SPs which don't occur in the sample of the *Expert*, we use the uninformative prior which is the probability of those patterns in the whole dataset. We repeat the approach with 5 and 10 expert samples from real dataset. Then, we apply the chi-square test to find if these 3 synthetic datasets, generated based on 1, 5 and 10 expert samples, are significantly different. The result of the test shows that we cannot reject the null hypothesis that these 3 synthetic datasets are from a common distribution. Thus, the difference between them is not significant. We repeat this process with sample of *Novice* developers to generate synthetic *Novice*s. To explore the scalability of the model, we generate 600 *"Expert"* and 600 *"Novice"* developers.

### 5.2 Fitting Zipf's law

According to [19], the maximum likelihood method should be used to estimate $\alpha$. Suppose we have $n$ syntax patterns: $\{sp_1, sp_2, ..., sp_n\}$ and that $Pr(sp; \alpha)$ is the probability distribution function of SPs. If the probability of SP, $sp_i$, is $Pr(sp_i; \alpha)$ then the maximum likelihood $L(\alpha)$ is defined as a joint probability of each $sp_i$:

$$L(\alpha) = \prod_{i=1}^{n} Pr(sp_i; \alpha) \tag{3}$$

We assume that all $n$ data points are independent and maximize the log-likelihood of $L(\alpha)$ [2, 14]. We can replace $\log Pr(sp_i; \alpha)$ with (2) in the log-likelihood function and estimate the parameter $\alpha$ in Zipf's law with (4):

$$\log L(\alpha) = -\alpha \sum_{i=1}^{n} f_i \log r_i - \sum_{i=1}^{n} f_i \log H_{n,\alpha} \tag{4}$$

To evaluate the goodness of fit, we apply Kolmogorov-Smirnov (KS) test and use the KS table of power law distribution [19].

The optimum sample size to fit a distribution with Zipf's law is based on Zipf [45], who showed that saturated Harmonic series $n * H_n$ can predict the optimum sample size in a Zipf's law (where in our study $n$ is the number of distinct SPs). Thus, the optimum size in our case study with 197 unique SPs is 1156 data points per each developer.

## 5.3 Comparable Methods

In this section, we discuss how we build different baselines and methods to investigate the accuracy of our approach. We first explain how these methods represent the knowledge of developers and then describe their technique to identify experts.

*Represent the Knowledge.* As a first baseline, we build a feature-based baseline inspired by previous works. Due to the fact that our approach to represent the knowledge of developers in a programming language is based on mastery of SPs, which is different from the case studies in past works (such as finding who is expert in specific libraries [31]), we could not directly compare our model to those proposed in these previous works. Hence, we decided to implement a new baseline using the features that were used in state of the art approaches from the literature.

A group of features that is commonly used in previous works is the *Quantity* of changes applied by developers such as the number of commits or lines of code [18, 25, 27]. The other group of features focuses on the *Frequency* of the activities such as the interval between two commit or recent dates of submission of a commit [3, 23]. The last group of features captures the *Breadth* of contributions using metrics such as the number of projects [12]. Authors in [28] calculate the correlations between different features to identify an expert. We replicate their model using the features summarized in Table 2. In the following, we refer to this model as baseline *"bl_ft"*.

**Table 2: features which are collect to represent the knowledge in Baseline *"bl_ft"***

| Feature Name | Description |
|---|---|
| numCommits | Number of commits in Python projects |
| LOC | Number of added and removed lines |
| avgInterval | Average interval between commits |
| lastCommit | Number of days since last commits |
| numProject | Number of Python projects that a developer contributed |

As a second baseline that is correlated with the study of Teyton et al [39], we use a vector space technique to represent the knowledge of a developer with the frequency of different SPs without fitting the distribution between them with Zipf's law. We calculate *tf-idf* of the frequency of SPs in developers' commits and then use them to represent their knowledge. This method helps us to check if representing the knowledge of developers with syntax patterns has any influence on identifying experts compared with simulating this knowledge with quantitative features such as the number of commits. We call this baseline *"bl_vs"*. As a last baseline, we build a naive model by representing the knowledge of each developer by the number of unique SPs in her/his commits and name it as *"bl_sp"*.

Due to the fact that the proposed mode, *Zipf_$\alpha$*, has single-dimensional data, we combine *"Zipf_$\alpha$"*, with *"bl_ft"* baseline. This hybrid method presents the impact of unifying quantitative features and the distribution of SPs on representing the programming knowledge of developers. To apply this combination, we add the expontent $\alpha$ as an extra feature into the list of features in *"bl_ft"* baseline.

*Classify Experts.* To identify experts, we apply both Clustering and Classification technique similar to what we explained in Section 4.4 for *Zipf_$\alpha$* approach. In clustering setup, we use Jenks Natural Breaks Clustering (JNBC) [22, 26] for single-dimensional data, *bl_sp*. Alternatively, we apply k-means clustering algorithm for methods with multi-dimensional data points such as *bl_ft*. In our case study, we define $k = 2$, because we focus on two categories: *Experts* and *Novice*. In classification setup, we choose random forest or SVM depending on the number of dimensions for multi-dimensional data and for single-dimensional data, we define the optimum threshold.

Table 3 shows a list of all methods evaluated in this paper; including our novel methods and baselines.

**Table 3: A description of different methods using to identify experts**

| Method | Description |
|---|---|
| **bl_ft** | A baseline that uses features in Table 2 to represent the knowledge of developers. |
| **bl_vs** | A novel baseline. It uses the vector space of developers and the frequency of SPs in their commits (SP-Dev-Matrix). |
| **bl_sp** | A baseline that uses the number of unique SPs as an indicator of expertise. |
| **zipf_$\alpha$** | The proposed model of the paper. It uses the exponent $\alpha$ as a factor to identify experts. |
| **zipf_$\alpha$_ft** | Combination of *zipf_$\alpha$* and *bl_ft*. |

## 5.4 Assessing the Performance of the Models

*Cross Validation.* We use Leave-one-out Cross-Validation (LOOCV) to evaluate the predicted categories in both setups. To use LOOCV in clustering techniques, each time we separate a single developer as a testcase and apply the clustering algorithm on the remaining developers as a trainset. We consider the cluster with maximum number of experts as an "Expert" cluster. Then, we calculate the distance between the feature(s) of testcase and the centroid of each cluster and predict the proper category by choosing the minimum distance.

*Evaluation Metrics.* We evaluate results using precision, recall, and F1-score for each category of *Expert* and *Novice*, separately. Further, we report Kappa statistic considering both categories.

## 5.5 Results

*RQ1: How does expertise impact the distribution between SPs in developers' commits?*

As we discussed earlier in Section 4, two developers with two different knowledge states choose different sets of SPs to write a code to solve the same problem. Writing different artifacts such as commits generates a distribution between SPs in the knowledge state of a developer. This distribution follows Zipf's law. We noticed that to fit different distributions with Zipf's law, different values are assigned to the exponent $\alpha$ in Zipf's law.

To answer the research question *RQ1*, we fit the generated data of developers in two categories of *"Expert"* and *"Novice"* with Zipf distribution. In other words, we separately fit the distribution of SPs in commits of each developers with Zipf's law and we obtained the value of exponent $\alpha$ per developer. Then, we visualize the distribution of $\alpha$ in two categories of *"Expert"* and *"Novice"*. Fig. 6 presents this distribution in both categories. We find that there is no intersection between the range of $\alpha$ for developers in these two categories. It shows the proficiency of developers at SPs influences the range of $\alpha$ to fit the distribution of SPs in thier commits. The *mean* of $\alpha$ is equal to 0.75 between *"Novice"* developers while it is equal to 0.96 between *"Expert"* ones.

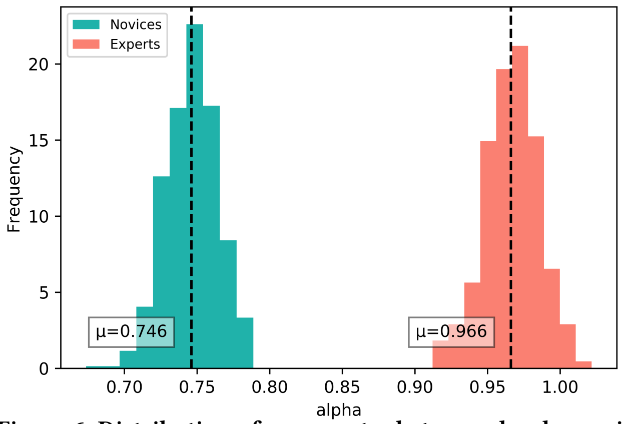


**Figure 6: Distribution of exponent $\alpha$ between developers in synthetic Data**

To find the optimum number of developers to be manually labeled from the real dataset, we conduct a hypothesis testing to find this optimum number of samples. According to Fig. 6, which shows different *mean* for $\alpha$ in two categories of *"Expert"* and *"Novice"*, we define a hypothesis as follow and apply a t-student test with 95% confidence to find the optimum size of data:

$H_0$: *"The means of distributions of exponent $\alpha$ are equal between novices and experts"*

$H_a$: *"The means are different between novices and experts"*

Based on the result of the test in Table 4, the optimum number of developers to reject the $H_0$ is equal to 20. Thus, we manually classified 20 *"Expert"*s and 20 *"Novice"*s.

*Manual Classification:* Two authors acting as "expertise evaluators" independently [28, 36, 39] performed an inspection of 20 *"Expert"* and 20 *"Novice"* developers from dataset of real developers. One of the evaluators is a Ph.D. candidate in software engineering, and the second is a master student, also in SE, both having a few years of professional experience. The evaluators checked different number of commits, pull requests and codes reviews submitted by developers in GitHub python projects until they felt confident in their assessments of developers. They inspected 10%, 20% or up to 30% randomly sampled of developers' contributions who have more than 500 contributions in their GitHub profile. For those who have less than 500 contributions, the evaluators checked 40%, 70% or up to all commits of developers. Also, they checked individuals' profile on Stackoverflow.com if they have one and read random samples Q/&A in which they are involved with the same procedure (as GitHub platform) for those who have more or less than 500 reputation in stackoverflow. In addition, they checked individuals profile in Linkedin.com if they could find them with their name and photos, to check their careers and list of skills. To ensure that both evaluators had a same perspective of the manual inspection, they performed two different procedures to categorize developers: the first evaluator classified developers in two categories of *Expert* or *Novice*, and the second evaluator scaled developers from *Novice* to *Expert* with labels of 1, $2^-$, $2^+$ and 3. The lowest level on the scale means: "she/he has contributed in python repositories (including individual repositories) but has no functional contribution such as code reviewing or adding functionality, optimization, testing or bug resolving commits in high star projects in Python repositories, or didn't answers questions related to python programming in Q/A platforms", and the highest level means: "she/he has different functional contributions in large Python projects, has answered questions about python programming and its libraries in Q/A platforms. Also, she/he may has related career background in python programming from Linkedin profile". Two middle scales, $2^-$, $2^+$, belong to individuals who are not acting as lowest/highest level, however they cannot be categorized in highest/lowest level. The team compared the given labels with given scales and calculate the percentage of agreement between the two evaluators using Cohen's Kappa[5]. They achieved 87.5% agreement between themselves. The disagreements are on 5 cases. For example, the Dev2 contributes in 8 Python projects with overall 1384 commits. He is an AI researcher and a postdoc student. First evaluator categorized him as "Novice" and second evaluator assigned him scale $2^+$. As another example, Dev5 has 316 commits in 36 projects. He is a python developer based on his Linkedin page but his contribution metric in GitHub is 43. First evaluator labeled him as "Expert" based on content of his commits. However, second evaluator scaled him as $2^+$. We keep disagreement cases and we calculate all evaluation metrics in *RQ2* twice, each time based on the manual inspection of one of the evaluators, and report their average.

After this labeling step, we fit the distribution of SPs in commits of these developers with Zipf's law. Fig. 7 shows the distribution of $\alpha$ for the selected sample of 40 real developers. Similar to our observation in synthetic data, we find that the mean of $\alpha$ is different in these two categories, despite several intersection between the range of $\alpha$.

**Table 4: hypothesis testing to find the optimum number of developers**

| t-statistic | p-value | sample size |
|---|---|---|
| -45.76 | <<0.0001 | 20 (20 novices and 20 experts) |

[5]http:// vassarstats.net/kappa.html

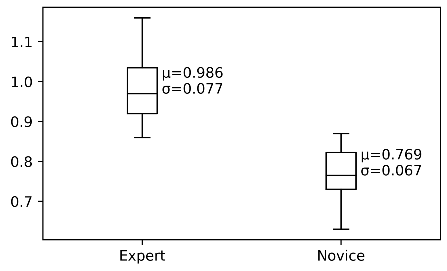


**Figure 7: Distribution of exponent $\alpha$ between developers in real data**

Answer to RQ1: *Syntactic mastery influences the distribution between Syntax Patterns (SPs) in commits of developers and exponent $\alpha$ in Zipf's law is an indicator of this mastership.*

*RQ2: How well can Zipf's law assess developers' mastery in programming SPs and discriminate experts from novices?*

In our second research question, we want to find how is the performance of our proposed approached compared with baselines.

Table 5 presents results in both clustering and classifier setups. The results for methods with single-dimensional feature, *zipf_$\alpha$* and *bl_ps*, are the same in both groups of clustering and classifier. The performance of our proposed method, *zipf_$\alpha$*, is greater than the performance of all the baselines. The precision, recall, and F1-score of our proposed method is 89.74%, 85.48% and 87.5%, respectively in both setups, when classifying *Experts*. The method *zipf_$\alpha$_ft* that is built by combining the quantitative features and the content of changes (mix of *zipf_$\alpha$* and *bl_ft*) has the highest performance in the classifier setup with precision, recall and F1-score of 92.5%, 90.39% and 91.4%, respectively.

The feature-based baseline, *bl_ft*, and the vector space baseline, *bl_vs*, shows a greater performance than the naive baseline, *bl_sp*, in both setups. It shows that the number of distinct SPs is not a good indicator of developers' knowledge by itself.

Methods with multi-dimensional features, *bl_ft*, *bl_vs* and proposed method *zipf_$\alpha$_ft*, show a better performance in Classifier setup. As an illustrative example, Dev8 which is categorized as an "Expert" (or scale 3) by both evaluators, is predicted wrongly as a "Novice" in clustering setup but correctly as an "Expert" with the classification technique. Dev8 has 1083 commits, contributed in 16 Python projects, his last commit was 26 days before our data collection date. He added 10499 code lines in different projects during the last year. He is one of the main developer of a 36k-star repository in GitHub. After applying k-means, the distance between the feature vector of this developer and the centroid of "Expert" and "Novice" cluster is 0.35 and 0.33 respectively in the normalize dataset of the *bl_ft* method. Thus, Dev8 is categorized in the cluster with minimum distance of 0.33. As another example, Dev15 is wrongly classified as a "Novice" with *bl_ft* in classifier setup. However, both evaluators agreed to classify him as "Expert". With the *zipf_$\alpha$_ft* method, in the classifier setup, he is correctly classified as an "Expert". This result shows that combining the syntactical patterns with quantitative features such as the number of commits improves the performance of a binary classifier to identify experts in programming.

Answer to RQ2: *Our proposed method, "zipf_$\alpha$" achieves a greater precision than other baselines methods to identify "Expert", in both Clustering and Classifier setups. The combination of quantitative features and the distribution of SPs as zipf_$\alpha$_ft method, led to performance improvements up to* 18.45%.

## 6 THREATS TO VALIDITY

The accuracy of estimating exponent $\alpha$ is influenced by two factors: "Number of SPs" and "Size of Corpus". Besides, the performance of our model depends on the "Validity of Data in GitHub". In this section, we discuss how these factors pose threats to the validity of our experiments.

### 6.1 Number of SPs and Size of Corpus

We focused on Python programming language in the case study we conducted. The number of SPs after applying AST on commits of developers in real dataset is 197. It can be different in other programming languages. Not all developers use all these 197 SPs in their code. The quality of estimating parameter $\alpha$ in Zipf's law depends on minimum number of SPs and the corpus size.

To show the dependency to the minimum number of SPs, we generate 4 synthetic datasets with Zipf distribution while $\alpha = 1, 0.9, 0.8$ and $0.7$, the corpus size ($T^* = 1156$) and the size of SPs equal to 197. Then, we calculate a confidence interval for each value of exponent $\alpha$. To do so, for each distribution, for example, $\alpha = 0.9$, we select a random sequence of SPs with the bootstrap method (with replacement) and generate 100 different sample datasets from the main population. Then, we estimate $\alpha$ for all 100 sample datasets and calculate the confidence interval of $\alpha$. To find the uncertainty of the model in estimating $\alpha$ as a function of the number of unique SPs, we vary the number of SPs from 197 to 5 and estimate the parameter $\alpha$ in the new distribution. Fig. 8 shows the estimated $\alpha$ in different number of SPs. For example, for $\alpha = 0.9$, if the number of SPs (or rank in Zipf's law) is less than 66, the predicted $\alpha$ is out of the confidence interval. We can see that for the number of SPs less than 63 (on average), the estimated $\alpha$ is out of the confidence interval.

Another factor that can impact the estimation of exponent $\alpha$ is the corpus size. It is the total number of SPs with frequency in the commits of developers. As discussed in Section 5.1, $n \cdot H_n$ indicates the optimum corpus size. The optimum corpus size in our case study with 197 distinct SPs is $T^* = 1156$. The Residual Standard Error (RSE), is less than 1% for a $Corpus_{size} > 1156$.

In our case study, our proposed model cannot make a reliable decision if a developer represents less than 63 distinct SPs in her/his commits, or a corpus size less than $T^* = 1156$. This is a kind of cold start issue for our method. To get around this issue, we exclude developers with fewer than 63 distinct SPs or less than 1156 corpus size.

### 6.2 Validity of Data in GitHub

In our dataset, we collect the contribution of developers in public GitHub projects. However, there are developers in GitHub who have sparse public contributions but stronger contributions in private projects. We don't have access to the private projects, while they could have provided the missing data for those developers with which we face the cold start issue. Also, the availability of this

**Table 5: results on real data in two setups of clustering and classifier**

| | Clustering | | | | | Classifier | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Proposed methods | | Comparable Methods | | | Proposed methods | | Comparable Methods | | |
| **metric/method** | **zipf_$\alpha$** (JNBC) | **zipf_$\alpha$_ft** (k-means) | **bl_ft** (k-means) | **bl_sp** (vocab size) (JNBC) | **bl_vs** (k-means) | **zipf_$\alpha$** (threshold) | **zipf_$\alpha$_ft** (random forest) | **bl_ft** (random forest) | **bl_sp** (vocab size) (threshold) | **bl_vs** (SVM) |
| **Precision%(Exp)** | **89.74** | 89.68 | 67.93 | 47.37 | 68.12 | 89.74 | **92.5** | 70.5 | 47.37 | 77.5 |
| **Precision%(Nov)** | 85.48 | **86** | 66.25 | 45.24 | 69.81 | 85.48 | **90** | 72.14 | 45.24 | 75 |
| **Recall%(Exp)** | 85.48 | **85.6** | 70.83 | 43.93 | 73.33 | 85.48 | **90.36** | 75.6 | 43.93 | 75.71 |
| **Recall%(Nov)** | 89.74 | **89.74** | 63.16 | 48.68 | 64.08 | 89.74 | **92.24** | 66.58 | 48.68 | 76.84 |
| **F1-score%(Exp)** | **87.5** | 87.37 | 69.32 | 45.58 | 70.54 | 87.5 | **91.4** | 72.95 | 45.58 | 76.59 |
| **F1-score%(Nov)** | 87.5 | **87.62** | 64.62 | 46.89 | 66.71 | 87.5 | **91.09** | 69.23 | 46.89 | 75.9 |

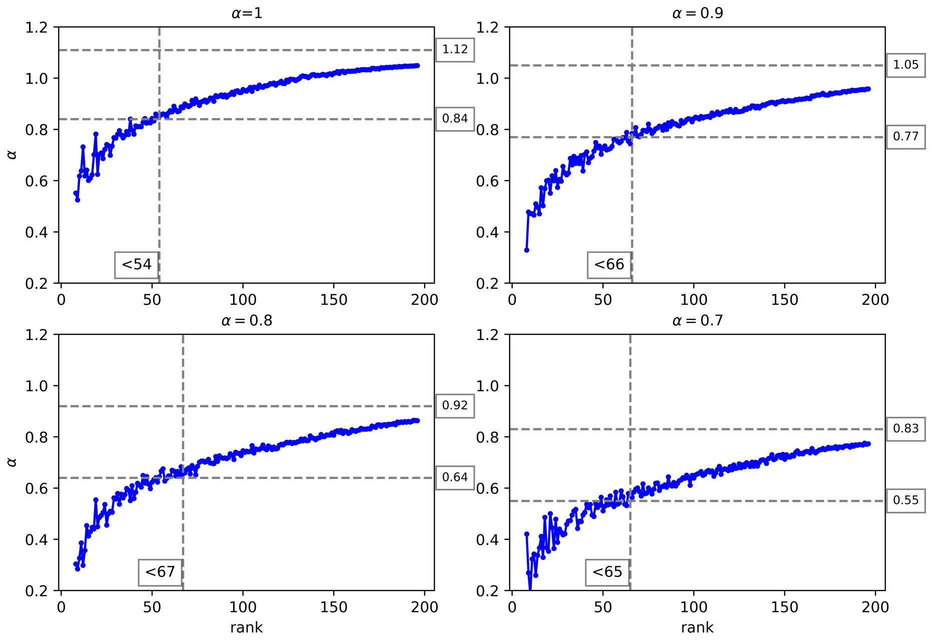


**Figure 8: Minimum number of SPs to fit Zipf's law and the confidence interval of $\alpha$ in $\alpha = 1, 0.9, 0.8$ and $0.7$**

information from private repositories could affect the decision of our method on cases in the *"Novice"* category.

Another threat to validity comes from the fact that we assume that all commits are written by the authors who have their name mentioned as being the author of the commit, and we did not considered the potential effect of code copying or code generative tools in the commits of developers.

External validity threats concern the possibility to generalize our results. Although our work focused on python projects, our proposed approach can be easily replicated for other programming languages. We provide all our data and scripts [6] to allow for a full reproduction and replication of our results.

[6]https://github.com/ExpertiseModel/ZipfModel

## 7 CONCLUSION

To improve the automatic identification of experts across different software project repositories in GitHub, we introduce a novel approach to represent the knowledge of developers in programming based on the Syntax Patterns (SPs) that they mastered and assess syntactic proficiency based on the distribution of these SPs in their commits. Our analysis shows that this distribution follows Zipf's law and the developers' syntactic mastery is reflected on the parameters of this distribution. To study the area of validity of the proposed model and discuss its sensitivity to different initialization, we generate synthetic data. To assess the effectiveness of our proposed model in identifying experts in programming, we conducted a case study with data from real developers. We compared the performance of our model with the performance of different state of the arts, with two group of Clustering and Classifier algorithms. Results show that our proposed model outperforms the state of the art approaches for the task of binary classification of programming experts. Also, the combination of our model with a baseline model from the literature led to a performance improvement of up to 18.45%.

In future work, we aim to extend our approach by collecting more advanced patterns as knowledge of developers. Also, we want to define patterns that specify the domain expertise of programmers. Finally, we are curious to study how to define different levels of expertise for developers in a specific knowledge domain.

## REFERENCES

[1] Mohammad Allahbakhsh, Boualem Benatallah, Aleksandar Ignjatovic, Hamid Reza Motahari-Nezhad, Elisa Bertino, and Schahram Dustdar. 2013. Quality control in crowdsourcing systems: Issues and directions. *IEEE Internet Computing* 17, 2 (2013), 76–81.

[2] Eduardo G Altmann and Martin Gerlach. 2016. Statistical laws in linguistics. In *Creativity and universality in language*. Springer, 7–26.

[3] John Anvik and Gail C Murphy. 2007. Determining implementation expertise from bug reports. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 2–2.

[4] Ali Sajedi Badashian. 2016. Realistic bug triaging. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 847–850.

[5] Jaume Baixeries, Brita Elvevåg, and Ramon Ferrer-i Cancho. 2013. The evolution of the exponent of Zipf's law in language ontogeny. *PloS one* 8, 3 (2013), e53227.

[6] Younes Boubekeur, Gunter Mussbacher, and Shane McIntosh. 2020. Automatic assessment of students' software models using a simple heuristic and machine learning. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 1–10.

[7] Martin S Bressler. 2014. Building the winning organization through high-impact hiring. *Journal of Management and Marketing Research* 15 (2014), 1.

[8] Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. 2019. Studying the difference between natural and programming language corpora. *Empirical Software Engineering* 24, 4 (2019), 1823–1868.

[9] Xiang Cheng, Shuguang Zhu, Gang Chen, and Sen Su. 2015. Exploiting user feedback for expert finding in community question answering. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*. IEEE, 295–302.

[10] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. 2009. Power-law distributions in empirical data. *SIAM review* 51, 4 (2009), 661–703.

[11] Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. 2010. Code comparison system based on abstract syntax tree. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. IEEE, 668–673.

[12] Jose Ricardo da Silva, Esteban Clua, Leonardo Murta, and Anita Sarma. 2015. Niche vs. breadth: Calculating expertise over time through a fine-grained analysis. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 409–418.

[13] Nilesh Dalvi, Anirban Dasgupta, Ravi Kumar, and Vibhor Rastogi. 2013. Aggregating crowdsourced binary ratings. In *Proceedings of the 22nd international conference on World Wide Web*. 285–294.

[14] Anna Deluca and Álvaro Corral. 2013. Fitting and goodness-of-fit test of non-truncated and truncated power-law distributions. *Acta Geophysica* 61, 6 (2013), 1351–1394.

[15] Tapajit Dey, Andrey Karnauch, and Audris Mockus. 2020. Representation of Developer Expertise in Open Source Software. *arXiv preprint arXiv:2005.10176* (2020).

[16] Roderick Edwards and Laura Collins. 2011. Lexical frequency profiles and Zipf's law. *Language Learning* 61, 1 (2011), 1–30.

[17] Jean-Claude Falmagne, Mathieu Koppen, Michael Villano, Jean-Paul Doignon, and Leila Johannesen. 1990. Introduction to knowledge spaces: How to build, test, and search them. *Psychological Review* 97, 2 (1990), 201.

[18] Thomas Fritz, Jingwen Ou, Gail C Murphy, and Emerson Murphy-Hill. 2010. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 385–394.

[19] Michel L Goldstein, Steven A Morris, and Gary G Yen. 2004. Problems with fitting to the power-law distribution. *The European Physical Journal B-Condensed Matter and Complex Systems* 41, 2 (2004), 255–258.

[20] Gillian J Greene and Bernd Fischer. 2016. Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 804–809.

[21] Weizhi Huang, Wenkai Mo, Beijun Shen, Yu Yang, and Ning Li. 2016. CPDScorer: Modeling and Evaluating Developer Programming Ability across Software Communities.. In *SEKE*. 87–92.

[22] George F Jenks. 1967. The data model concept in statistical mapping. *International yearbook of cartography* 7 (1967), 186–190.

[23] Jennifer Marlow and Laura Dabbish. 2013. Activity traces and signals in software developer recruitment and hiring. In *Proceedings of the 2013 conference on Computer supported cooperative work*. 145–156.

[24] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. 2009. Assigning bug reports using a vocabulary-based expertise model of developers. In *2009 6th IEEE international working conference on mining software repositories*. IEEE, 131–140.

[25] David W McDonald and Mark S Ackerman. 2000. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. 231–240.

[26] Robert McMaster. 1997. In Memoriam: George F. Jenks (1916-1996). *Cartography and Geographic Information Systems* 24, 1 (1997), 56–59.

[27] Audris Mockus and James D Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 503–512.

[28] Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Identifying experts in software libraries and frameworks among GitHub users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 276–287.

[29] João Eduardo Montandon, Marco Tulio Valente, and Luciana L Silva. 2021. Mining the Technical Roles of GitHub Users. *Information and Software Technology* 131 (2021), 106485.

[30] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[31] Johnatan Oliveira, Markos Viggiato, and Eduardo Figueiredo. 2019. How Well Do You Know This Library? Mining Experts from Source Code Analysis. In *Proceedings of the XVIII Brazilian Symposium on Software Quality*. 49–58.

[32] Regina Pustet. 2004. Zipf and his heirs. *Language Sciences* 26, 1 (2004), 1–25.

[33] Ali Sajedi-Badashian and Eleni Stroulia. 2020. Vocabulary and time based bug-assignment: A recommender system for open-source projects. *Software: Practice and Experience* (2020).

[34] David Schuler and Thomas Zimmermann. 2008. Mining usage expertise from version archives. In *Proceedings of the 2008 international working conference on Mining software repositories*. 121–124.

[35] Evgeny Shulzinger, Irina Legchenkova, and Edward Bormashenko. 2018. Co-occurrence of the Benford-like and Zipf Laws Arising from the Texts Representing Human and Artificial Languages. *arXiv preprint arXiv:1803.03667* (2018).

[36] Renuka Sindhgatta. 2008. Identifying domain expertise of developers from source code. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 981–989.

[37] Leif Singer, Fernando Figueira Filho, Brendan Cleary, Christoph Treude, Margaret-Anne Storey, and Kurt Schneider. 2013. Mutual assessment in the social programmer ecosystem: An empirical investigation of developer profile aggregators. In *Proceedings of the 2013 conference on Computer supported cooperative work*. 103–116.

[38] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 908–911.

[39] Cédric Teyton, Marc Palyart, Jean-Rémy Falleri, Floréal Morandat, and Xavier Blanc. 2014. Automatic extraction of developer expertise. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. 1–10.

[40] Yuan Tian, Dinusha Wijedasa, David Lo, and Claire Le Goues. 2016. Learning to rank for bug report assignee recommendation. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

[41] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. 2020. An empirical study of c++ vulnerabilities in crowd-sourced code examples. *IEEE Transactions on Software Engineering* (2020).

[42] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. 2015. Dual analysis for recommending developers to resolve bugs. *Journal of Software: Evolution and Process* 27, 3 (2015), 195–220.

[43] Hongyu Zhang. 2009. Discovering power laws in computer programs. *Information processing & management* 45, 4 (2009), 477–483.

[44] George Kingsley Zipf. 1949. Human behaviour and the principle of least-effort. Cambridge MA edn. *Reading: Addison-Wesley* (1949).

[45] George Kingsley Zipf. 2016. *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books.