

Learning from Examples to Improve Code Completion Systems

Marcel Bruch, Martin Monperrus, and Mira Mezini
Software Technology Group, Darmstadt University of Technology
{bruch,monperrus,mezini}@st.informatik.tu-darmstadt.de

ABSTRACT

The suggestions made by current IDE's code completion features are based exclusively on static type system of the programming language. As a result, often proposals are made which are irrelevant for a particular working context. Also, these suggestions are ordered alphabetically rather than by their relevance in a particular context. In this paper, we present intelligent code completion systems that learn from existing code repositories. We have implemented three such systems, each using the information contained in repositories in a different way. We perform a large-scale quantitative evaluation of these systems, integrate the best performing one into Eclipse, and evaluate the latter also by a user study. Our experiments give evidence that intelligent code completion systems which learn from examples significantly outperform mainstream code completion systems in terms of the relevance of their suggestions and thus have the potential to enhance developers' productivity.

Categories and Subject Descriptors

D.2.6 [Software Engineering]: Programming Environments

General Terms

Algorithms, Documentation

1. INTRODUCTION

The code completion feature of modern integrated development environments (IDEs) is extensively used by developers, up to several times per minute [21]. The reasons for their popularity are manifold. First, usually only a limited number of actions are applicable in a given context. For instance, given a variable of type `java.lang.String`, the code completion system would only propose members of this class but none of, say, `java.util.List`. This way, the code completion prevents developers from writing incompilable code by proposing only those actions that are syntactically correct. Second, developers frequently do not know exactly which method to invoke in their current context. Code completion systems like that of Eclipse use pop-up windows to present a

list of all possible completions, allowing a developer to browse the proposals and to select the appropriate one from the list. In this case, code completion serves both as a convenient documentation and as an input method for the developer. Another beneficial feature is that code completion encourages developers to use longer, more descriptive method names resulting in more readable and understandable code. Typing long names might be difficult, but code completion speeds up the typing by automating the typing after the developer has typed only a fraction of the name.

However, current mainstream code completion systems are fairly limited. Often, unnecessary and rarely used methods (including those inherited from superclasses high up in the inheritance hierarchy) are recommended. Current code completion systems are especially of little use when suggestions are needed for big (incoherent) classes with a lot of functionality that can be used in many different ways.

For illustration, consider the public interface of the class `SWT1Text`, which consists of more than 160 callable methods. Whenever querying the system for instances of type `Text` an overwhelming number of proposals are made—including all methods of `java.lang.Object` (e.g., `equals()`, `notify()`, or `wait()`)! But actually methods like `wait()` are never invoked on an instance of `Text` from within the whole Eclipse codebase—after all, a software with several millions of lines of code. Recommending this method every time is rarely helpful to a developer, thus, unnecessarily bloats the code completions and counteracts the (last two) advantages of code completion systems.

The method `wait()` is not an isolated phenomenon. By analyzing the Eclipse codebase, we found that typically no more than five methods are invoked on `SWTText` instances. Thus, a developer needs to pick the right 5 method calls from the list of 160 proposals. Even with Eclipse's capability to narrow down the list of proposals with each new character typed by the developer, the list still remains unnecessarily large.

Callable methods differ not only with respect to the frequency of their use; they also often differ with respect to the specific contexts in which they are typically used. Several factors such as the current location in code, e.g., whether the developer is currently working in the control-flow of a framework method [6], or the availability of certain other variables in the current scope affect the relevance of a method.

For illustration, consider the situation of a developer creating a dialog window to gather user input using a text widget. Typically, text widgets have a two-phase life-cycle: (a) they are configured and placed in a visual container during dialog creation and (b) they are queried for user input after the dialog window is closed. These two phases are typically encoded in different methods of the dialog.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ESEC-FSE'09, August 23–28, 2009, Amsterdam, The Netherlands.
Copyright 2009 ACM 978-1-60558-001-2/09/08 ...\$10.00.

¹a graphical user interface library

While widget configuration takes place within `Dialog.create()`, reading the input occurs within `Dialog.close()`. Depending on the dialog method within which the developer needs suggestions for method calls on a text widget, different methods are relevant. Within `Dialog.create()`, relevant methods are text widget creation methods and setter methods for its visual properties; within `Dialog.close()` the `getText()` method is relevant.

The suggestions made by mainstream code completion systems are based exclusively on the information given by the static type system of the programming language. This seems too primitive given that the examples discussed above suggest that taking into account factors like the frequency of certain method calls in certain contexts might help to improve the accuracy of the proposals made by code completion systems.

Our hypothesis for this paper is as follows: The quality of suggestions and hence the productivity of software development can be improved by employing *intelligent code completion systems* capable of

1. filtering those elements from the list of proposals which are irrelevant for the current context, thus, disburdening a developer from knowing all (unnecessary) details of the API used, instead allowing him to focus only on API elements that are actually relevant.
2. assessing the relevance of every proposal (e.g., by using a relevance ranking), thus allowing a developer to quickly decide which recommendations are relevant for the task at hand

In this paper, we propose *intelligent code completion systems* that learn from existing code repositories by searching for code snippets where a variable of the same type as the variable for which the developer seeks advice is used in a similar context. We have built three prototype code completion engines of this kind, each using the information contained in repositories in different ways. The first prototype uses the frequency of method calls as a metric to decide about their relevance. The second code completion uses association rule mining to search the code repository for frequently occurring method pairs. Finally, the last and most advanced code completion system recommends method calls as a synthesis of the method calls of the closest source snippet found. To build this system, we have modified the *k nearest neighbors* algorithm [8], a classical and efficient machine learning algorithm, to fit the needs of code completion. We call the resulting algorithm *best matching neighbors (BMN)* algorithm.

To evaluate our hypothesis we use a large scale evaluation process for recommender systems, sketched in [6], along with standard information retrieval performance measures. By large scale, we mean that the system is evaluated with a test bed of more than 27,000 test cases. The evaluation results prove the efficiency of the learning code completion engines in general and of the *best matching neighbors (BMN)* algorithm in particular. In order to demonstrate that the *best matching neighbors (BMN)* code completion algorithm, which gets the best quantitative evaluation, has the potential to increase developer productivity, we show how it can be seamlessly integrated into the default Eclipse development widgets and evaluate its usefulness by means of a user study.

The remainder of this paper is organized as follows. In Sec. 2 we present learning algorithms for code completion systems under investigation and especially our main contribution: the *best matching neighbors (BMN)* algorithm. Sec. 3 presents the setup and results of the quantitative evaluation of the learning code completion systems which are compared against each other and against the default Eclipse code completion system. Sec. 4 presents the prototype

implementation of the BMN algorithm and its integration into the Eclipse IDE. Sec. 5 shows the results of a user study that used our tool in real-life situations. In Sec. 6 we give an overview of related work in this area and this paper concludes with Sec. 7.

2. EXAMPLE-BASED CODE COMPLETION

The Eclipse code completion system (EcCCS for short) uses the type of a variable and suggests all callable method names based on this information. It only needs the information about the type hierarchy. This system serves as baseline for more intelligent code completion systems (CCS for short). Clearly, since it proposes all possible method names, the "correct" methods are always among its suggestions. The question is how many irrelevant recommendations are also made.

Our hypothesis, which we will test in the evaluation section, is that EcCCS makes too many irrelevant recommendations, since it does not take into consideration the context in which a particular object is used. This motivates our work on more intelligent code completion systems that learn how to use objects of a particular type in a particular context from code other developers have written in similar situations.

2.1 Three New Code Completion Systems

We have implemented three code completion systems that learn from existing example code. These systems use different kind of information as the basis for completing calls.

A frequency based code completion system.

A plausible approach for intelligent CCS is to determine the relevance of each method based on the frequency of its use in the example code. The rationale for this strategy is: The more frequently a method has been used the more likely it is that other developers will use the same method again. By implementing and evaluating such a frequency based code completion system (FreqCCS for short), we want to find out to which extent such a rather simple frequency based relevance ranking system can help developers to find the right completions.

An association rule based code completion.

Association rule mining is a machine learning technique for finding interesting associations among items in the data [2]. The problem of mining association rules is to find all rules $A \rightarrow B$ that associate one set of items with another set. Association rules have already been used in the context of recommendation systems for software engineering. Codeweb [20] generates usage pattern documentation. Fruit [5] makes interactive usage recommendations when developing software using frameworks. Both approaches do not handle recommendations on variable level. To the best of our knowledge, there is no publication that proposes to apply association rules in the context of code completion systems.

However, it is possible to use an example codebase to mine variable-scoped association rules and to use the context of a variable for determining the rules to select. In a nutshell, association rules can be used for code completion system.

Consider the introductory example of using a `Text` widget again. The data mining process of association rules may identify two different usages: 1) Object creation which involves a constructor call and calls to several setter methods like `setText()`, and 2) object interrogation, when the object is queried for its state by calling getters like `getText()`. An association rule would be then "If a new instance of `Text` is created, recommend `setText()`". Another rule would be "If in `Dialog.close()`, recommend `getText()`".

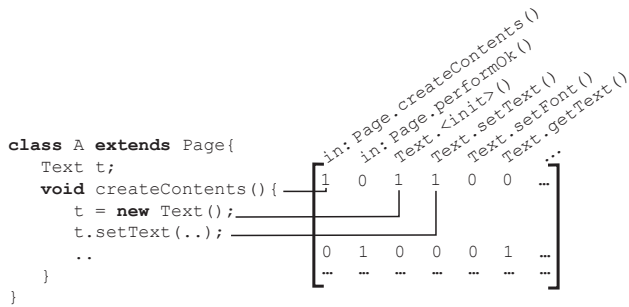


Figure 1: Encoding Framework Usages as Binary Vectors

We have implemented such an association rules based code completion system (ArCCS for short).

The Best Matching Neighbors code completion.

This new code completion system is based on a modification of the k-nearest-neighbor (kNN) machine learning algorithm [8]. To the best of our knowledge, kNN has not been used so far for code completion system. Hence, this algorithm is the main contribution of this paper. For this reason, we present this code completion engine in detail in the following section.

2.2 The Best Matching Neighbors Completion System

In this section we present the *best matching neighbors algorithm* (BMN for short). BMN adapts the k-nearest-neighbor (KNN) algorithm [8] to the problem of finding method calls to recommend for particular objects.

The KNN algorithm comes from the pattern recognition research. It is a classification algorithm. For instance, in the context of image recognition, given the image of a letter, the algorithm predicts the letter. The intuition of the algorithm is based on a common sense rule which can be enounced as follows: to predict something according to some observation, let's find in one's experience a similar situation, and predict what actually happened at the end.

The KNN algorithm fits remarkably well to the problem of code completion: when a developer wants to complete code at the usage pattern level, she might start to search for code fragments very similar to the already written ones (e.g. using Google Code) and takes inspiration from the rest of the code. In a nutshell, our algorithm works as follows:

1. Extract the context of the variable;
2. Search for variables used in similar situations in an example codebase;
3. Synthesize method recommendations out of these nearest snippets.

More specifically, given a local variable v in the code under development, the system extracts and encodes the context as a feature vector. Based on this information, the algorithm searches the example base for object usages that are *close* to the usage being codified. From the close examples, the algorithm recommends the methods that are most likely to be used. The BMN code completion system (BMNCCS for short) is a tailoring of the KNN algorithm to the context of code completion. Our tailorings are:

1. The way of extracting the context of the variable and encoding it as a feature vector;

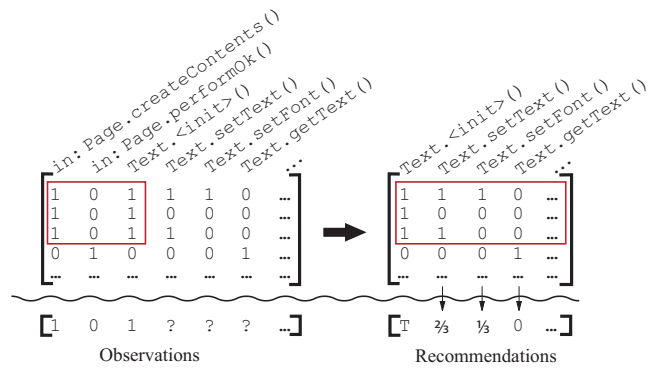


Figure 2: From Observations to Recommendations

2. The design of meaningful and efficient distance measure between the code being written and the snippets of the example code base;
3. The selection mechanism of nearest neighbors;
4. The synthesis of method recommendations out of these nearest snippets.

2.2.1 Codebase Variables as Binary Vectors

The BMN system is built for completing code that uses a particular framework.

Given a local variable t in the code the developer is working on, whose type comes from the framework under consideration, the BMN system extracts the declared type of the variable, the names of the methods already invoked on t , the method within which t is being used. Then it encodes them as binary information.

Figure 1 illustrates this extraction and encoding process: the feature *Text.setText()* is set to 1 if *setText()* is called on t , and the feature *in:page.PerformOK()* is set to 1 if the variable t is used in the method context *Page.PerformOK()*.

Each variable of the example codebase is extracted from the example codebase using the Wala bytecode toolkit² and encoded as described above. This process ends up with an example code base represented by a set of feature vectors, i.e. a binary matrix.

Let k be the number of variables extracted, i be the number of method contexts found in the training code base, and j be the number of all callable methods on all framework types. Then the usage binary matrix U has k rows and $i + j$ columns.

Since a variable can be in one method context only, the matrix U has the following property: for each observed usage v , there is a single context feature $f_t, 0 < t \leq i$ such that $v_f = 1$, i.e. each usage happens in the context of exactly one method; Such a usage binary matrix is shown at the left hand side of figure 2.

2.2.2 A Distance Measure for Code Completion

The KNN algorithm default distance measure is the Euclidean distance. First, we observe that since we are in a binary feature space, the Euclidean distance is exactly the square root of the Hamming distance³, as proved in the following equation.

²<http://wala.sf.net>

³The Hamming distance between two feature vectors is the number of positions for which the corresponding features are different.

$$\begin{aligned}
euclidean(u, v) &= \sqrt{\sum_{i=1}^n (u_i - v_i)^2} \\
&\text{since } u_i, v_i \text{ are 0 or 1} \\
&= \sqrt{|u_1 - v_1| + \dots + |u_n - v_n|} \\
&= \sqrt{\sum_{i=1}^n diff(u_i, v_i)} \\
&= \sqrt{hamming(u, v)}
\end{aligned}$$

where $diff(u_i, v_i) = 1 \iff u_i \neq v_i$

This observation leads to a high-performance implementation of the code completion system since the calculation of Hamming distance is efficient (in comparison to the calculation of Euclidean distance).

Second, the KNN algorithm default distance measure is based on the full feature space. We show in the following that it does not fit to code completion.

The top matrix in the left part of figure 2 is an encoding of an example code base as feature vectors. The bottom left corner of the figure shows a context (observation) given to the code completion system. The observation vector at the bottom left corner of figure 2 encodes the situation when the code completion is triggered in the context of overriding the method `Page.createContents()` and after an instance of `Text` is created. The ones and the zeros in the context vector denote facts we are sure about. For instance, for the observed vector of figure 2, we are sure that we are within a method that overrides `createContents()` and that we are not within a method that overrides `performOK()`. However, we cannot say for sure whether the developer does not want to use `setText()`, or whether she simply has not yet used it, or whether she does not even know about its existence. Question marks in the vector that encode the context denote uncertainty.

Using a distance on the whole feature space requires treating uncertainty as zeros. It turns out that in a real world feature space there are much more uncertain feature values (i.e., question marks in figure 2) than certain feature values. In such a case, the Hamming distance captures only the noise due to uncertainty. We encountered this problem empirically. This can be explained as follows using probabilities.

Let us model the terms of the Hamming distance, $diff(u_i, v_i)$, as a random variable following the Bernoulli distribution, which has the following characteristics:

$$\begin{aligned}
E(diff(u_i, v_i)) &= p \\
var(diff(u_i, v_i)) &= p \cdot (1 - p)
\end{aligned}$$

Let us now consider the variance of the Hamming distance by splitting its terms in two groups: related to certain and uncertain information. Let also assume that these two groups are uncorrelated, then thanks to the Bienaymé formula, we can write:

$$\begin{aligned}
var(hamming(u_i, v_i)) &= var\left(\sum_i diff(u_i, v_i)\right) \\
&= var\left(\sum_{i \in certain} diff(u_i, v_i)\right) \\
&\quad + var\left(\sum_{i \in uncertain} diff(u_i, v_i)\right)
\end{aligned}$$

which, thanks to the central limit theorem, can be transformed to:

$$\begin{aligned}
var(hamming(u_i, v_i)) &= \sqrt{n_{certain} \cdot p \cdot (1 - p)} \\
&\quad + \sqrt{n_{uncertain} \cdot p \cdot (1 - p)}
\end{aligned}$$

Since $p(1 - p)$ is a constant and $n_{uncertain} \gg n_{certain}$ in our feature space, the Hamming distance is driven in this context by $n_{uncertain}$, i.e., captures only the noise of uncertainty. Practically, this means that using the KNN algorithm on the whole feature space gives poor results for code completion. Our solution to this problem is to compute the distance on a partial feature space, based only on certain information of the observed context.

The first modification made to the initial KNN algorithm is to compute the distance based on *certain information* only. Given an incomplete vector (`iCompVec`) encoding the context, the algorithm iterates through all rows of the binary usage matrix and determines the distance between each row and `iCompVec`, based on the columns of `iCompVec` that contain certain information. This is illustrated in figure 2, where only the three first columns of the matrix are used to compute the distance between the context and the example codebase. Eventually, there are three snippets close to the observation at an equal Hamming distance of 0: the three first rows of the example code base (marked by a red rectangle).

To sum up, the BMN system computes the distances between the current programming context and the example codebase based on the Hamming distance on a partial feature space.

2.2.3 The Selection Mechanism of Nearest Neighbors

In a standard pattern recognition context, when using the KNN algorithm, features are real numbers. Hence the distance between the observation and the database is also a real number that allows a complete ordering of neighbors.

In the context of code completion and with the distance measure described in the previous section, it turns out that a lot of neighbors are at the same distance of the input vector. It means that there is no complete ordering of the neighbors and it does not make sense to select the K nearest neighbors, whatever K is. It is very likely to have other neighbors that are exactly equally distant to the input observation. For instance, in the example of figure 2, there are three equally close neighbors.

To address this problem, we build equivalence classes based on the calculated distance and then take the set with the smallest distance. This set contains best matching neighbors and ground the name of the algorithm. This set is used to compute the method call recommendations.

2.2.4 Synthesizing Recommendations

Since the KNN algorithm is a classification algorithm, it retrieves a class (for instance 'A', 'B', etc. in the context of letter recognition).

The BMN algorithm acts differently. Once the nearest snippets are selected, it computes the likelihood of the missing method calls based on their frequency in the nearest snippets, i.e., by counting the occurrence of each method call in the selected snippets and dividing it by the total number of selected snippets.

In the example in figure 2, the nearest snippets of the context are the three first rows. The BMN algorithm then looks at the methods called in each snippet and counts their occurrence. For instance, the method `Text.setText()` occurs in two out of the three records and `Text.setFont()` occurs only once. Method `Text.getText()`, however, is never observed, whenever the constructor call has been observed. When dividing these numbers by

the total number of selected rows, we get the following recommendations: $\frac{2}{3}$ of the closest examples called `Text.setText()`, $\frac{1}{3}$ called `Text.setFont()`, and none of the examples matching the current observation called `Text.getText()`.

The BMN algorithm is parameterized with a threshold, called a filtering threshold. If the likelihood of a method call is higher than the filtering threshold, the methods are recommended and ordered by their likelihood. In the example of figure 2, considering a threshold of 50%, the BMN code completion system recommends only one method call to `setText()`; the method call has a likelihood of 66%.

To sum up, the BMN algorithm identifies method calls to be recommended to the user based on their frequencies in the *selected* nearest neighbors.

3. EVALUATION

In the following, we evaluate the Eclipse code completion and the three code completion systems (CCS) introduced in Sec. 2.

3.1 Evaluation Data Set

We measure the ability of the code completion systems under investigation to predict method calls on objects of the Standard Widget Toolkit (SWT), a graphical user interface library [9]. The SWT library has been chosen for several reasons:

- There are an important number of open source programs that uses SWT. These programs constitute the "knowledge" base from which BMN, FreqCCS, and ArCSS systems can learn.
- Also many frameworks rely on the SWT framework. Developers must know the concepts of both frameworks, e.g., where to place the method calls to SWT instances in the context of the other framework etc. As discussed in the introductory example, the knowledge about the current overridden framework method can be leveraged by a code completion system to improve the proposals.
- SWT is used in many projects, hence a code completion system for it could be immediately beneficial to a great audience of developers.

We collected the whole Eclipse 3.4.2 codebase for conducting the experiment. This codebase grounds both the code completion system and the creation of the evaluation scenario. To allow future comparison with other approaches, we will make this dataset public available on the project homepage.

3.2 Evaluation Scenario

The code completion systems under investigation are evaluated with several thousands of queries following the automated evaluation process presented in [6]. In the following, the main steps of the evaluation process and their rationale are briefly summarized.

1. The initial data set is randomly split in two parts: 90% grounds the initial knowledge of the code completion system, the remaining 10% are called test data and are used to create evaluation scenario. This step ensures that the system is not evaluated with queries for which it has the *exact* information to answer them. This is a crucial requirement for evaluating machine learning systems [4].
2. For each binary feature vector of the test data, some method calls (i.e., some 1s) are removed. The resulting degraded vector will be used as a query to the system. The removed method calls constitute an expectation, similar to the expected

<pre>1 Text t = new Text(..); 2 t.</pre>	<pre>1 Text t = new Text(..); 2 t.setText(..); 3 t.setLayoutData(..); 4 t.setFont(..);</pre>
(before)	(after)

Figure 3: Code Snippet Before & After Code Completion

value of a unit test case. This step simulates a real programming situation, where the developer needs assistance after she has already written some code. We remove half of the method calls of the initial feature vector thus simulating the situation where the developer has already performed half of the job. This choice is a pragmatic trade-off between: (a) providing information that enable to make intelligent context-dependent predictions, and (b) withholding information from the system to complicate the task of prediction.

The dataset we used for this evaluation contained more the 27,000 examples usages, and the number of method calls in these examples varied from 2 up to 46 methods calls.

The randomization in step #1 ensures that, on an average, the query distribution between small and hard tasks reflects the real distributions of such situations in the codebase.

3. For each query, each code completion system is asked to return a prediction of the missing method calls.
4. For each prediction and the corresponding expectation, evaluation metrics defined in Sec. 3.3 are calculated.

To achieve a standard machine learning evaluation, we repeat the evaluation process 10 times; this is known as 10-fold cross validation [18]. Since the initial dataset consists of 27,000 records, the code completion systems are actually evaluated against $(27,000 * 0.1) * 10 = 27,000$ real world programming queries.

3.3 Evaluation Metrics

The objective of this evaluation is to figure out to which extent the four code completion systems under investigation, EcCCS, FreqCCS, ArCCS, and BMNCCS, can: (i) identify relevant methods, i.e., methods that are actually used at the end by the developer to complete her code, and (ii) weed out irrelevant proposals, i.e., those that did not make it into the final code, without sifting out too many of the relevant methods.

The measures we use for assessing the performance are precision, recall, and the F1-measure. These measures are commonly used for evaluating the performance of information retrieval systems like (web) search engines or generally any kind of recommender systems. The intelligent code completion systems that are subject of this evaluation are basically recommender systems in a particular domain.

To explain the meaning of the metrics consider the code snippet depicted in Listing 3. This listing shows an incomplete usage of a `Text` widget on the left hand side and the final code after as it should be. Assume that a developer that actually knows exactly what methods to call on `t` nevertheless pressed *Ctrl-Space* in line 2 on the code snippet on the left hand side to query the code completion system. Assume further, that the system returned three recommendations: `setText()`, `setLayoutData()` and `addListener()`. The developer investigates these recommendations and decides to apply the first two proposals, to ignore the third one, and to add a new, not recommended call to `setFont()`.

Let us measure the performance of the CCS for this query. The developer added three method calls to the code. Two calls were predicted correctly by the system (`setText()`, `setLayoutData()`); the third call (`setFont()`) was not proposed. Thus, the system proposed 2/3rd of the calls the developer actually needed. This is exactly the recall the CCS achieved for this query! More formally, recall is defined as the ratio between the relevant (correct) recommendations made by the system for the given query and the total number of recommendations that it *should* have made:

$$Recall = \frac{Recommendations_{made \cap relevant}}{Recommendations_{relevant}}$$

Clearly, a system that recommends all possible methods would always achieve a great recall. But, if only one out of a hundred recommendations is actually needed, the system is not really intelligent. This is where the second interesting question, namely, how many false recommendations the system made, comes into place. In our example, 2/3rd of the recommendations actually made it into the final code. This is the precision that the CCS achieved for the query. Formally, precision is defined as the ratio between relevant recommendations made and the *total* number of recommendations made by the system for a particular query:

$$Precision = \frac{Recommendations_{made \cap relevant}}{Recommendations_{made}}$$

Both values together allow to assess the quality of a CCS for a given query. In order to summarize how the systems perform on several queries the precision and recall values are averaged over all queries[1]. After averaging, the performance of each completion system is boiled down to a pair of numbers. However, with two numbers characterizing the "goodness" of systems, the question arises when a system is "better" than another. It might be the case that one system has a very high recall but a very low precision. Another system might have only a medium recall but also a medium precision. The F-Measure[26] has been widely accepted by the research community as a means to correlate precision and recall by computing their harmonic means:

$$F = \frac{(1 + \beta^2) \cdot precision \cdot recall}{\beta^2 \cdot precision + recall}$$

The F-Measure allows to emphasize either recall *or* precision by accordingly assigning the β parameter. For our evaluation, we equally weight precision and recall ($\beta = 1$). The resulting formula is called the F1 measure [26].

In addition to summarizing the performance in a single number, the F1-Measure has another important role in our evaluation. Each CCS under evaluation has a set of specific parameters (e.g., minimum likelihood thresholds etc.) that affect its performance. To produce comparable results we need to minimize the effects of accidental parameter guessing. For this purpose, we used the F-Measure as an optimization criterion for each CCS: We run several dozen experiments for each CCS to figure out the best parameter settings that maximized the F-Measure.

3.4 Evaluation Results

Figure 4 summarizes the results of the evaluation. It shows the recall, precision and F1 levels reached by each code completion system. As already mentioned, all systems were tuned to maximize F1 with the corresponding algorithm parameters (e.g. the filtering threshold of FreqCSS and BMN), except EcCCS which has no parameter to be tuned.

The EcCCS performs worst. As expected, it proposes all relevant methods (i.e., it has a recall of 100%), but its low precision shows

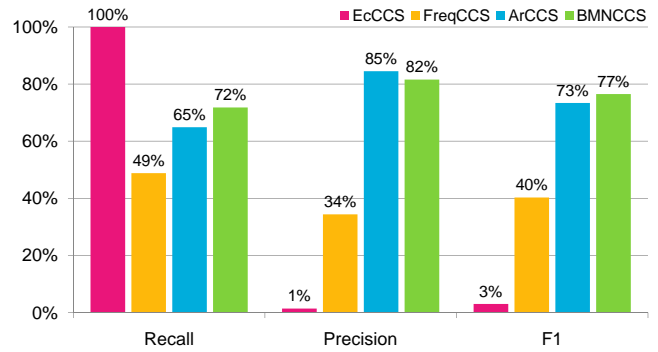


Figure 4: Performance of EcCCS, FreqCCS, ArCCS and BMN

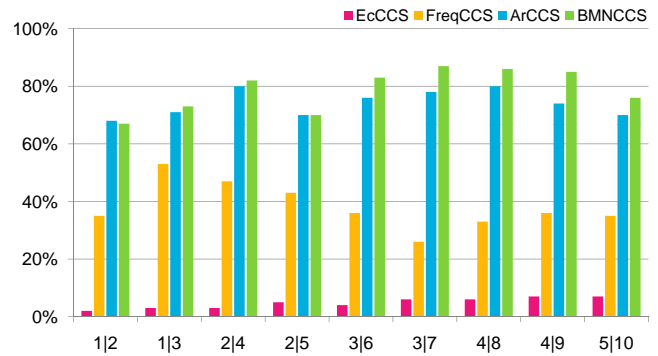


Figure 5: F1 per Expected Method Call Recommendations

that 99% of its recommendations were not what the developer actually needed, thus false proposals in our setup. This result clearly suggests that current code completion systems provide a large room for improvements.

The frequency-based CCS achieves significantly higher precision than the EcCCS; yet, its overall performance is disappointing. Only one out of three proposals was actually correct. Furthermore, the system only finds half of all relevant method calls. Even if the underlying idea would seem intuitive, we doubt, that code completion systems purely based on the frequency of method calls would meet the developer expectations.

The rule-based approach significantly improves both values: It finds 65% of all relevant methods and, what is really interesting, 85% proposals were correct. This shows that the proposals made by the ArCCS are highly reliable and developers can trust recommendations made by this system.

Finally, the BMNCCS achieves a 10% higher recall than ArCCS and a slightly lower precision. In total, from all evaluated code completion systems the BMN system achieves the highest F1 value, thus performs best with respect to this evaluation setup.

Figure 5 enables us to refine these conclusions by decomposing the F1 performance along the different query settings. The results are grouped by the total size of method calls contained in the example snippets. For illustration of this figure, consider the second group of bars labeled with "1|3". The bars summarize the performance of the code completion systems for the following queries: Given one randomly selected method call, the system was asked to return the two missing calls; in total the example snippet contained 3 calls. This figure lets us draw the following conclusions.

First, the overall performance of the evaluated systems stays in the same order of magnitude. Since the F1 measure stays around

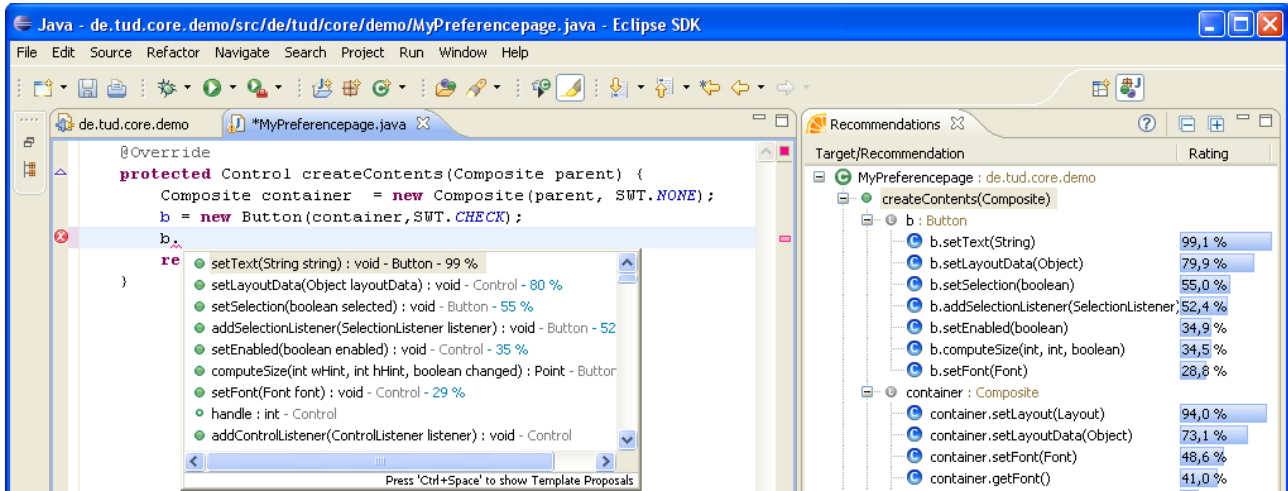


Figure 6: Integration of our Intelligent Code Completion into Eclipse

80%, we can say that the BMN code completion system is able to predict method calls based on large usage patterns involving dozens of method calls. Second, the gap between ArCCS and BMN widens when the completion task size increases. While ArCCS and BMN are roughly equivalent for predicting method calls based on short usage patterns, BMN is significantly better for large usage patterns.

To sum up, the best code completion according to this evaluation is the BMN system. It finds 72% of all relevant method calls and 82% of the recommended method calls are actually correct. The performance results of this new system give encouraging evidence that intelligent code completion systems are not a utopia.

3.5 Discussion

In the following, we discuss the the generalizability of the evaluation results.

- Our evaluation deals with Eclipse code, i.e. our context (encoded as feature vectors) contains information about the object-oriented context of a variable. For instance, the context might say that the current variable is in a class that extends `DialogPage`, and in a method that overrides a specific method of `DialogPage`, say `setControl`. This additional information may improve the performance of the system, and weaken the generalizability of the very high precision and recall.
- Our evaluation deals with SWT objects. It seems that SWT classes contain more logic, more complex usage patterns than classes of the default Java API (e.g. `java.net.URI`). Again, the generalizability of the very high precision and recall might be discussable for low-level APIs, such as default Java API. Currently, we are conducting further experiments to assess the performance of the system on different target libraries (without object-oriented contexts, and without complex usage patterns).
- Evaluation queries are built by randomly selecting method calls from real variables of the codebase, without taking into account the order. This may lead to unrealistic queries. For instance, if one considers the ordered calls on a variable, we may randomly drop out the middle third of the calls. In this case, it is unlikely that a developer would write code this way and that the code completion system would be queried with

this context. We are in the process of obtaining empirical results to figure out: 1) whether our approximation increases or decreases the overall precision or recall and 2) to what extent the order of method calls matters (for instance, methods that configure an object may be called in different orders).

4. INTEGRATING BMN INTO ECLIPSE

In this section, we present our prototype integration of the BMN engine into the Eclipse IDE. We decided to integrate BMN since it was the engine with the best performance according to the evaluation. The goal for building the prototype was to enable the user study presented in Sec. 5 to give us insight on the practical relevance of code completion systems capable of learning from example code. We present the prototype by elaborating on three main differences between of our prototype to the default Eclipse code completion system. Our statements are illustrated by figure 6, which shows a screenshot of our prototype.

Size of the method name list.

First of all, an advanced code completion system substantially reduces the number of methods suggested to those with high probability of being relevant in a particular context. We agree with Robbes and Lanza[22] that we cannot expect the programmer to scroll down a huge list of method names in order to find the good one. This would introduce a cognitive context switch in the programmer activity.

Our system filters recommendations based on a threshold on the confidence value of the recommendations. The higher the threshold, the lower the number of recommendations in the code completion widget. Based on our experience in using earlier versions of the prototype, the default filtering threshold is set to 30%.

Confidence value.

The default code completion widget is enhanced with the confidence values of the recommendations (for instance, consider in the screenshot the 99% confidence value of method `setText`). This value is an indicator for the developer of what to do next. We use the following guidelines to interpret the confidence values.

A very high confidence value ($\sim >90\%$) indicates that unless the developer explicitly knows that she is implementing a limit case, this method has to be called.

A high confidence value ($\sim >50\%$) means that this method can probably be called. From the programmer view point, the interpretations can be:

1. “I know the usage pattern I am implementing, I know this method and I am happy the tool agrees with me”. *The tool may strengthen the confidence the programmer has in her knowledge and in her code.*
2. “I did not plan to use this method. May be I am wrong in using this class. Let’s read the corresponding method documentation”. *Our code completion system can be used as a knowledge watchdog. The difference between the developer’s plan and the confidence value is a kind of warning.*
3. “Hey, what is this method? I don’t know it. Let’s read the corresponding method documentation, it can be an important one.” *Our code completion system helps the programmer to discover new features. It assists her in improving her knowledge.*

A low confidence value ($\sim <50\%$) is a reminder of potential methods to call in special usage patterns. Since special usage patterns are rarely used, the programmer forgets them easily. In this case, our code completion system can act as a reminder.

Note that the thresholds for interpreting the confidence values are fuzzy. When one gets used to work with a code completion system with confidence values, each programmer tends to tune these values w.r.t. the framework used and to her experience. Note also that the validity of these guidelines depends highly on the application of the single responsibility principle in the class being used. If a class can be used in several different ways, i.e., it violates the single responsibility principle, the confidence values computed by our code completion system automatically decrease and the thresholds loose their relevance.

Dedicated view.

We propose to add a new view to Eclipse, dedicated to code completion, shown on the right hand side of figure 6. In contrast to the code completion window (which shows recommendations for a single variable only) this view provides a summary of all recommendations (including their relevance) available for the given class and its variables. This view serves as a browser, allowing a developer to search through all recommendations and supports her to understand the instance usage concepts.

Conclusion.

Our new code completion system is tightly and seamlessly integrated into the Eclipse IDE. It does not break the usual programmer way of working: 1) advanced code completion is still available with `Ctrl-Space` (the default keyboard shortcut that every Eclipse programmer knows) 2) code completion still works without noticeable delays (i.e., computing the distances between the current context and the codebase is not an issue in terms of performance). The improvements compared to the existing Eclipse code completion widgets are: 1) irrelevant methods are removed 2) method recommendations are always given together with a confidence value and 3) we introduce a new view dedicated to code completion.

5. USER STUDY

In Sec. 3 we demonstrated the effectiveness of *intelligent code completion systems* in identifying relevant methods for a given context using a large-scale automated evaluation approach. In this section, we report on the user feedback of 10 experienced Java programmers who tested the BMN system for its usefulness. The user

study consists of three phases: (i) completing a predefined task, (ii) filling a questionnaire after the programming task, and (iii) giving an interview one or two days after returning the questionnaire.

5.1 Setup

For the user study each subject had to develop a small graphical user interface using the Eclipse SWT UI Framework. This user interface required several graphical SWT widgets like `Buttons`, `Text fields`, `Combo boxes` etc., as well as some interactions between these widgets in response to some user interactions like enabling and disabling of widgets etc. The appearance of the interface and the dynamic behavior was specified with an annotated screencast.⁴ Furthermore, we provided the basic application code as a downloadable Eclipse project to free developers from Eclipse-specific tasks unrelated to the user study as such.

Overall, ten subjects participated in the user study. All subjects had several years of experience with the Java programming language and were familiar with Eclipse and its code completion system. Half of the subjects did not have any experience with the SWT framework; the other half had at least several months of experience in developing SWT applications and, thus, were acquainted to the concepts of the SWT framework.

5.2 Results

For the questionnaire and interviews we asked the subjects to answer several questions concerning the usefulness of the proposed system and to give their personal rating to these questions. For the personal ratings the subjects had to choose one of *Strong Agree* ($++$), *Weak Agree* ($+$), *Weak Disagree* ($-$), *Strong Disagree* ($--$), or *No Answer* (\circ) if none of the previous was deemed appropriate. The significant questions from the questionnaire are given below (Questions 2 and 5 are summaries from interviews as they were not rating-based questions).

1. Did the system propose relevant method calls? ($5++ / 4+ / 1\circ$) This question aims to identify whether subjects’ perception was in alignment with the numerical results of the automated evaluation process. Nine out of ten subjects were pleasantly surprised about the quality of the recommendations made by the system but some criticized that in few cases the system also recommended unrelated methods, which lead to deductions in the rating.

2. Did the system correctly rank the proposals by relevance? (Interview) This question aims to identify how valuable the subjects found the two ways for presenting relevance of a recommendation to the users supported by the system, namely, (a) by presenting the likelihood along with the proposal and (b) by ranking the recommendations by the identified relevance. Most developers stated that they largely ignored the probability of a proposal and just followed the order of the proposals on the code completion window. The interviews suggest that displaying the exact relevance (percentage) is not a necessary feature as long as the most relevant recommendations are found on top of the list.

3. Did the tool speed up your development compared to the default Eclipse code completion? ($4++ / 5+ / 1-$) One reason for the success of code completion is that it speeds up coding. This question aims to catch the subjective perception whether intelligent code completion systems can *further* improve development speed compared to the default Eclipse code completion. We take the obtained feedback as a positive indicator of the usability of the tool that shows its potential for future research.

⁴<http://www.stg.tu-darmstadt.de/research/core/>

4. Is the tool well integrated into Eclipse? (7++ / 1+ / 2--)

The aim of this question is (i) to identify how pleased the subjects were with the implementation, and (ii) whether some issues with the interface existed that might affect other aspects of the tool. The user feedback suggests that the integration into the Eclipse's code completion is well accepted by the users. Two subjects complained that the Javadoc for the SWT framework was missing, thus, reducing the usability of the tool. Since this is probably due to a misconfiguration of the Eclipse system itself, these complains do not diminish the general positive feedback about the integration. Based on this feedback, we conclude that the survey results were not distorted by implementation issues.

5. Did the tool help you to understand the concepts of the framework? (Interview) Since the intelligent CCS recommends likely method calls, one hypothesis was that the tool also might support developers in understanding the framework concepts faster. The interview showed that this hypothesis is currently unfounded. Usually, framework concepts have an abstraction level much higher than method calls, thus, presenting likely methods did not help the novice subjects of this user study to grasp the underlying framework concepts. Furthermore, the subjects identified several other issues with learning a framework which we will elaborate on in Sec. 7.

Summarizing, the promising results of the user study show that it is reasonable to assume that developers would accept intelligent code completion systems as a valuable extension to the toolbox available for modern software engineering.

6. RELATED WORK

A large number of framework documentation techniques exists [7]. Prescriptive techniques such as cookbooks, tutorials, exemplars, and pattern languages document how to customize a framework in order to accomplish a specific task [11, 15]. Other work is directed to the documentation of a framework's architecture, so that users understand the rationale behind the design [3]. While helpful, these techniques require that the developer puts much effort to browse and find the good tips concerning the code being written. In contrast, with our approach, the documentation generation is fully automated and integrated into the development environment; the generated documentation is context-sensitively presented to the user by the code completion system.

Code Templates provide a (pre-coded) means of inserting larger code snippets, such as for the creation of an SWT Button. Although code templates are powerful, they suffer from the problem that their creation and maintenance is costly; consequently only few common patterns are encoded using templates. Furthermore, a large number of framework interactions are context-sensitive and consist of single method calls only. For these cases typically no templates exist.

Another category of tools enable users to find code snippets in code repositories. Several tools treat code like regular text documents and apply information retrieval technology to find code examples [10, 12, 17].

Software exploration tools like Sextant [24] or modern IDEs like Eclipse provide queries to search for code elements based on structural properties. While these search features are better than documentation browsing, they still require that the developer is involved in some heavy interactions with the tool.

Some tools feature implicit queries, i.e., the user is no more required to write a query, but the latter is generated automatically from the current context. For instance, Codebroker [27] uses signature information and comments to actively present similar program elements in its repository. Strathcona [14] uses the structural con-

text of the code under development, e.g., the super type, method calls, or overridden methods to find examples with a high similarity to this context. One disadvantage of example-based tools is, that the developer still has to do the mapping between the recommended code snippet and its own current programming task. Intelligent code completion systems address a different problem. Rather than proposing example code, they use examples to identify common usages of the framework and assess the relevance of method calls found in common usages for the developers current context, giving a developer an intuition which methods are actually relevant and which ones she might want to ignore.

Mylyn [16] shows methods that have been frequently and recently interacted with by the developer at the top of the content assist list. Our approach is complementary. Mylyn learns what to put at the top based on the developer's personal usage history; intelligent code completion systems proposed here learn what to put on top by analyzing code submitted by other developers.

There are three noticeable approaches to help developers when using a framework. They all address the same problem statement: the developer knows what type of object is needed, but not know how to write the code to get the object. Prospector [19] addresses this issue by first creating elementary "jungloids" from framework methods that model the direct reachability of a type t_{out} given a type t_{in} . When the user searches for a type t , the tool follows the jungloids starting from the current code context (all accessible types) until t is reached and returns the code snippet modeled by the synthesized jungloid chain. XSnippet [23] and PARSEWeb [25] extend this approach by improving the ranking heuristics, the query capabilities, and the mining process, and, respectively, by using a code search engine as an example repository which enables to collect code samples on demand. Compared to these approaches, we address a different problem: given an object of some type, what methods can be called on it in a certain context. It is to be noted that Prospector, XSnippet, and Parseweb were evaluated with at most 40 different tasks, while the performance of BMN is assessed by more than 27000 test cases.

Hill and Rideout presented an automatic method completion system [13] which addresses the issue of writing so-called atomic clones. Atomic clones are small scale and good *copy/paste* of source code. Hill and Rideout argue that atomic cloning is a useful development practice: transforming an atomic clone as a reusable unit is too heavy weight and hinders the design and the extensibility of the software. In order to both help the developer and speed up the writing of an atomic clone, the KNN algorithm is used to match in a code base an atomic clone that is then directly inserted in the source code. While the KNN algorithm and the Euclidean distance matches the problem of searching for atomic clones, we showed why and how to improve the KNN algorithm to create a good code completion system that helps the developers to choose methods to call on an object.

Robbes and Lanza [22] showed that it is possible to leverage the program history to improve the quality of code completion. They define and evaluate 6 new algorithms for method name completion. Our approach differs by important points. *Evaluation data*: Their evaluation strategy requires to have a complete change history of software, obtained by using specific plug-in of the IDE over years of software development. On the contrary, our evaluation requires only to have client source code of the framework or library under study without any special versioning/change data. Note that since we do not have such specific change data, we cannot quantitatively compare our algorithms to theirs. *Replicability*: The data they used for evaluation is not publicly available. On the contrary, our evaluation is replicable since we use the publicly available Eclipse code-

base. *Scope*: Their algorithms are defined and evaluated in the case where the programmer has already entered at least the two first characters of the method names. Our approach is more ambitious: we aim to predict relevant method names *without any prefix entered* and proved that it is possible.

7. CONCLUSION AND FUTURE WORK

In this paper, we introduced the concept of *example based code completion system*. These systems are *intelligent*, their knowledge is based on information mined in an example codebase. They improve the state-of-the-art of code completion systems by producing context-sensitive and relevant method call recommendations, while remaining seamlessly integrated into the IDE.

We presented three example based code completion systems. We conducted a large scale evaluation of these three systems with 27000 real world code completion queries extracted automatically from the example code base. We showed that these systems dramatically outperform the type-based Eclipse code completion system. The best system built is able to predict 82% of the method calls that are actually needed by the programmer (recall) and 72% of the recommended method calls are relevant (precision). It is based on a variant algorithm of the machine learning algorithm K-nearest neighbors, which we called Best Matching Neighbors (BMN).

We performed a user study involving 10 subjects to figure out whether real world developers could benefit from such a new code completion system. The results are promising: 9 out of 10 subjects think that an example based code completion system speeds up the development.

There are several areas for future work. Further experiments are needed to gain a more precise understanding of the generalizability of the evaluation results for other kinds of software systems such as classical libraries and standard APIs, as mentioned in Sec.3. Also, there is a lot of machine learning techniques that could leverage the example codebase to build a better method recommendation model. Also, a feedback that we received from the user study was, once you are recommended a relevant method by the system, you still have to figure out yourself how to handle parameters to the call, which is a demanding task. Our future work will address this issue. We believe that it is possible to build a system that recommends what to do with the parameters: using an existing object, instantiating a new one, using a constant value, etc. are all possible alternatives.

8. REFERENCES

- [1] *Overview of the Third Text REtrieval Conference (TREC-3)*, Gaithersburg, MD, USA, 1990. NIST.
- [2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Int'l Conf. Management of Data*, pages 207–216. ACM, 1993.
- [3] K. Beck and R. E. Johnson. Patterns generate architectures. In *ECOOP*, pages 139–149. Springer, 1994.
- [4] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [5] M. Bruch, T. Schäfer, and M. Mezini. FrUiT: IDE support for framework understanding. In *OOPSLA Workshop Eclipse Technology Exchange*, pages 55–59. ACM, 2006.
- [6] M. Bruch, T. Schäfer, and M. Mezini. On evaluating recommender systems for api usages. In *RSSE'08*, pages 16–20, New York, NY, USA, 2008. ACM.
- [7] G. Butler, R. K. Keller, and H. Mili. A framework for framework documentation. *ACM Computing Surveys*, 32(1):15–21, 2000.
- [8] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 1967.
- [9] Eclipse Foundation. SWT: The standard widget toolkit. <http://www.eclipse.org/swt/>, 2006.
- [10] W. Frakes and B. Nejme. Software reuse through information retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.
- [11] D. Gangopadhyay and S. Mitra. Design by framework completion. *Automated Software Eng.*, 3(3/4):219–237, 1996.
- [12] Google code search. <http://www.google.com/codesearch>.
- [13] R. Hill and J. Rideout. Automatic method completion. In *ASE*, pages 228–235, 2004.
- [14] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125. ACM, 2005.
- [15] R. E. Johnson. Documenting frameworks using patterns. In *OOPSLA*, pages 63–72. ACM, 1992.
- [16] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *FSE*, pages 1–11. ACM, 2006.
- [17] Koders. <http://www.koders.com>.
- [18] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1137–1145, 1995.
- [19] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61. ACM, 2005.
- [20] A. Michail. Data mining library reuse patterns using generalized association rules. In *ICSE*, pages 167–176. ACM, 2000.
- [21] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23(4):76–83, 2006.
- [22] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of ASE*, 2008.
- [23] N. Sahavechaphan and K. Claypool. Xsnippet: Mining for sample code. In *OOPSLA*, pages 413–430. ACM, 2006.
- [24] T. Schäfer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT software exploration tool. *IEEE TSE*, 32(9):753–768, 2006.
- [25] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE*, pages 204–213. ACM, 2007.
- [26] C. J. van Rijsbergen. *Information retrieval*. Butterworths, London, 1979.
- [27] Y. Ye, G. Fischer, and B. Reeves. Integrating active information delivery and reuse repository systems. In *FSE*, pages 60–68. ACM, 2000.