Title
Collective Intelligence for Smarter API Recommendations in Python

Permalink
https://escholarship.org/uc/item/9qj3z9j0

Author
D Souza, Andrea Renika

Publication Date
2016

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA,
IRVINE

Collective Intelligence for Smarter API Recommendations in Python

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE
in Informatics

by

Andrea Renika D’Souza

Thesis Committee:
Professor Cristina Videira Lopes, Chair
Associate Professor James A. Jones
Associate Professor Sam Malek

2016
DEDICATION

To my family and my friends for their unfailing support and continuous encouragement throughout this journey.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT OF THE THESIS</td>
<td>viii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivations</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Thesis Statement</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 Literature Review</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Static Analysis for Dynamic Languages</td>
<td>7</td>
</tr>
<tr>
<td>2.2 “Intelligent” Approaches to Recommend APIs</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Evaluation of Recommenders</td>
<td>11</td>
</tr>
<tr>
<td><strong>3 Approach</strong></td>
<td>13</td>
</tr>
<tr>
<td>3.1 Data Extraction</td>
<td>14</td>
</tr>
<tr>
<td>3.2 Analysis of Python source files</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Best Matching Object Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>3.3.1 Additional context information</td>
<td>19</td>
</tr>
<tr>
<td>3.3.2 Selection of Nearest Neighbors</td>
<td>20</td>
</tr>
<tr>
<td><strong>4 Evaluation</strong></td>
<td>22</td>
</tr>
<tr>
<td>4.1 Experiment Procedure</td>
<td>22</td>
</tr>
<tr>
<td>4.2 Experiment Dataset</td>
<td>23</td>
</tr>
<tr>
<td>4.3 Evaluation Metrics</td>
<td>23</td>
</tr>
<tr>
<td>4.4 JEDI</td>
<td>25</td>
</tr>
<tr>
<td><strong>5 Results and Findings</strong></td>
<td>26</td>
</tr>
<tr>
<td>5.1 Impact of Additional Context Information</td>
<td>26</td>
</tr>
<tr>
<td>5.2 Prediction with different types of queries</td>
<td>28</td>
</tr>
<tr>
<td>5.3 Comparison with JEDI</td>
<td>31</td>
</tr>
</tbody>
</table>
6 Conclusion

6.1 Threats to Validity ..................................................... 33
6.2 Future Work ............................................................. 34

Bibliography ................................................................. 36
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Examples of Dynamic behavior in Python</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Example code snippet</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Program Graph generated for code snippet in Figure 3.1</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Encoding the file objects as frequency vectors</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Best Matching Objects from extracted Object Usages</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>Code Recommendations using Nearest Neighbors</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>MRR for models with Additional Context Information</td>
<td>27</td>
</tr>
<tr>
<td>5.2</td>
<td>Recall for models with Additional Context Information</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>MRR for Different Query Types</td>
<td>29</td>
</tr>
<tr>
<td>5.4</td>
<td>Object Usages found for Different Query Types</td>
<td>29</td>
</tr>
<tr>
<td>5.5</td>
<td>Impact of number of object usages on evaluation metrics</td>
<td>30</td>
</tr>
<tr>
<td>5.6</td>
<td>Impact of number of object usages on query types</td>
<td>30</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparison of PyReco and JEDI results for re</td>
<td>31</td>
</tr>
<tr>
<td>5.8</td>
<td>Comparison of PyReco and JEDI results for django</td>
<td>31</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 Reaching Definitions and extracted context information stored in Nodes . . . 16
ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor and committee chair, Professor Cristina Videira Lopes for her help and guidance throughout the course of this project. Her honest feedback and insights were invaluable in formulating this thesis.

I am also grateful to my committee members, Professor James A. Jones and Professor Sam Malek for their time and support in reviewing this document.

I would like to thank to my research collaborator and current Ph.D. student, Di Yang. This thesis would not have been possible without the weekly brainstorming sessions with her.

I would like to extend a sincere thank you to all my fellow lab mates in Mondego for their insightful feedback during the workshop presentation.

Last but not least, I would like to thank all my students in ICS 31, for teaching me Python like no other MOOC or textbook could.

This work was partially supported by a grant from the DARPA MUSE program.
Software developers use Application Programming Interfaces (APIs) of libraries and frameworks extensively while writing programs. In this context, the recommendations provided in code completion tools help developers choose the desired methods. The candidate lists recommended by these tools, however, tend to be large, ordered alphabetically and sometimes even incomplete. A fair amount of work has been done recently to improve the relevance of these code completion results, especially for statically typed languages like Java. However, these proposed techniques rely on the static type of the object and are therefore inapplicable for a dynamically typed language like Python.

In this thesis, I present PyReco, an intelligent code completion system for Python which uses the mined API usages from open source repositories to order the results based on relevance rather than the conventional alphabetic order. To recommend suggestions that are relevant for a working context, a nearest neighbor classifier is used to identify the best matching usage among all the extracted usage patterns.

To evaluate the effectiveness of our system, the code completion queries are automatically extracted from projects and tested quantitatively using a ten-fold cross validation technique. The evaluation shows that our approach outperforms the alphabetically ordered API recommendation systems in recommending APIs for standard, as well as, third-party libraries.
1

Introduction

“I believe we must place much greater emphasis on what I would call “the learning society” - a society that learns to listen to itself, reflect on itself, and create new possibilities for its future”

– Richard Harwood, Focus on Study Circles

1.1 Background

Most programming languages are filled with several diverse Application Programming Interfaces (APIs). These APIs help software developers extend the functionality of their programs and improve software quality with very little additional code. However, most software developers spend most of their time learning how to use these APIs rather than enjoy the benefits of software reuse which these components provide.

Robillard, in his study with Microsoft developers [33], notes that most participants attribute this difficulty of learning APIs with the scarce amount of learning resources available. He
speaks about the need for software tools that could foster a much more efficient way to learn the different ways an API can be used and could also provide a rationale for its design.

Code completion is one of the most widely used features in Integrated Development Environments (IDEs). In a study conducted by Murphy et al. on 41 Java developers using Eclipse [27], this feature was used 70% of the time and as commonly as copy-paste. Stylos and Clarke discovered that developers use this feature with the main objective of writing code faster, more correctly and to explore APIs [35].

The auto-complete tool in IDEs uses the static type of an object to list out all possible attributes or methods that can be invoked and is usually triggered when a user types a “.” character. However, Măraoiu et al. [25] observed that a large fraction of the recommendations made by this feature were not being accepted by the users and this tool was being used more for debugging purposes.

Hence, recent research focuses towards building “intelligent” code completion tools which order the proposals based on relevance rather than the conventional alphabetical order. These tools aim to improve the developer’s productivity and software quality by capturing the goal of a user for generating the completion lists.

With its rapid development features, simple and readable syntax, Python’s popularity has been growing in the development community. Being a dynamic language, the traditional and intelligent techniques for code completion cannot be used for recommending APIs in Python. Variables in Python are not given a type; instead, they take the type of whichever object is assigned to them during run-time. This triggers the need for a new approach that shifts from the traditional static analysis to a technique that could intelligently recommend APIs for Python.
1.2 Motivations

Current code completion tools in Python have been ineffective for API recommendations due to the following reasons:

(1) **Long lists of recommendations**
Since code completion displays all the possible methods and attributes which can be invoked by an object, these lists tend to be very long. For example, auto-complete invoked for a String object in Python would retrieve around 85 results by JEDI [4].

(2) **Alphabetically ordered recommendations**
This large alphabetically ordered list makes it hard to navigate to the correct candidate, sometimes making it even slower than typing the full name of a method directly. Developers tend to rely more on prefix filtering than scrolling to reduce the number of choices [25].

(3) **Scarce or Ambiguous documentation on APIs**
API Documentations could be ambiguous, especially in explaining the type of the object returned by these APIs. This problem is complicated even further in Python since the language does not have static types defined.

For instance, the documentation for `urllib.open` states that this method would return a *file-like object*. Developers could find this description of the return type confusing since it does not give a clear indication of whether the methods or attributes of a *file object* can be invoked on the object returned by `urllib.open`.

(4) **Incomplete Static analysis on Libraries**
Static analysis tools generate stubs for libraries which are then used to assist the development tools in recommending API methods. PyCharm [8] uses python skeletons [5] whereas Mypy [9] is an optional static analysis tool that uses typeshed [10] for the
These above-mentioned skeletons contain definitions for some of the most commonly used libraries. The list of libraries, however, is not be complete since the generation of these stubs is highly dependent on the ability to perform static analysis on the source code of these libraries. According to Madsen [23], it is complicated to statically analyze libraries due to the following reasons:

(a) Software libraries may be partially or fully implemented in another programming language.

(b) The source code for libraries may be large and not available for static analysis.

(c) Dynamic features may be used in the source code of these libraries.

An absence of recommendations could, however, cause developers to suspect the presence of an error in the program or to check the additional documentation available and thus negatively impact their productivity [25].
Failure to detect Dynamic behavior

Current code completion tools in Python use type inference along with available documentation to approximate the type of an object and then recommend methods or attributes for it.

However, using Monkey Patching [6], APIs of modules can be added or changed during runtime. In Figure 1.1a, the built-in function `str` is modified to return a string object of type `unicode` instead of `str`.

In a dynamic language, it is possible that a variable can have a set of values at a particular point in the program. Some code completion systems may fail to recognize these “union types” for an object. For example, in the code snippet shown in Figure 1.1b, the type of object named `a` is guessed correctly to be both `int` and `str` in PyCharm [8].

1.3 Thesis Statement

The issues listed in Section 1.2 that exist in current code completion systems for Python sparks the need for new recommenders that have a greater understanding of developers’ goals and of Python’s dynamic behavior in order to suggest APIs that are more suited to the programming task.

In this thesis, we present PyReco, an API recommender that is based on the idea of using collective intelligence to deduce the relevant methods for code completion. In this approach, we use the API usage patterns extracted from open source repositories rather than rely on conventional type inference techniques for the purpose of code completion.

The intuition behind our approach is that the large number of extracted API usage patterns present in these projects should be able to capture all the diverse scenarios in which APIs are currently being used by developers.
We propose the following solutions to address the issues and challenges mentioned in Section 1.2:

(1) We have limited the number of recommendations to a maximum of ten in PyReco, to not overwhelm the user with long lists in the code completion pop-up.

(2) The recommendations are ranked based on relevance using a Nearest Neighbor classifier which identifies the closest matching API usage pattern in the extracted list. In our classifier, we use the call sequences and additional context information for training the recommender.

(3) The extracted usage patterns contain information on methods and attributes that have been used by developers in similar programming scenarios.

(4) With PyReco, we will be able to propose code recommendations for all the libraries and APIs that were used in the mined open source projects. Thus, the completeness of this list is based on the popularity of libraries among developers rather than the inability to do static analysis on library code.

(5) Our parsing approach leverages the semantics of the Python language and control flow information present in program to predict the dynamic behavior more accurately.
Literature Review

In this section, we review the related work in order to answer the following research questions that we plan to address in later sections of this thesis:

(1) How can static analysis be used to infer types for library objects in Python?

(2) How can we order API recommendations based on relevance?

(3) How can we evaluate API recommenders effectively using automated approaches?

2.1 Static Analysis for Dynamic Languages

Code completion and other IDE tools use the static type of an object to recommend methods and attributes for it. However, in a dynamic language, most decisions like the type of an object and control flow are deferred till run-time [17]. In Python, these dynamic introspective and reflective features allow a variable to have different types at the different points in a program [24]. In the following paragraphs, we will discuss the several attempts that have
been made to infer types for dynamic languages like Python at compile-time since it can be useful as a first step for code completion.

Aycock [13] considers flow insensitivity and type consistency for inferring types in Python. The key idea behind his approach is that “giving people a dynamically-typed language does not mean that they write dynamically-typed programs.” Though his approach works for Python to Perl conversion, ignoring control flow in the code may not work favorably for code completion and it could lead to inaccurate completion proposals being generated at different points in a program.

Maia et al [24] use AST trees for type inference for RPython which contains only a subset of Python’s dynamic features. However, we need to account the dynamic features in Python to improve on current completion systems.

Cannon in his thesis [17], uses flow control to infer types for atomic objects. His approach was developed to help code optimization and for checking illegal operations. He observed that “the control flow within a method or function call could be static and also that data is usually stored in objects, instead of being passed around in methods.” Thus, an inter-procedural analysis on Python source code could be less effective in predicting the dynamic behavior.

Madsen [23] in his analysis with JavaScript libraries notices that applying static analysis to libraries could be complicated due to various reasons (described in Section 2.2). To infer the methods that can be used on library objects, he proposes Use Analysis which is “a heuristic for recovering missing data-flow facts, due to missing library code, by observing how applications objects are used in application code.” His approach consists of analyzing the client code of APIs of these libraries to infer types. We think that this approach could be more useful to deduce the types than traditional type inference techniques on the source code of libraries.
2.2 “Intelligent” Approaches to Recommend APIs

In the past few years, there has been a fair amount of research to improve the relevance of API recommendations using context information, machine learning and statistical approaches.

Robbes and Lanza [32] propose a code completion tool which uses temporal information like the program history to provide more relevant completions. On similar lines, Lee et al. [22] have an additional temporal dimension for evolutionary information on the code. In a collaborative work environment, they think that such information could make development tasks easier.

The semantic or structural information in programs is most commonly used for context in recommenders. Heinemann et al. claim that the identifier could be a good indicator for the methods which can be called for the development task. For instance, an object named \texttt{angle} could indicate the relevance of suggesting \texttt{sine} and \texttt{cosine} operations. A context sensitive completion approach by Asaduzzaman et al. tokenizes semantic information like keywords, method, class or interface names from the preceding lines as part of the context to improve the relevance of the code completion results.

Hou et al. [20] use a combination of grouping, sorting and filtering techniques to improve code completion. In grouping, the APIs are grouped on the basis of their functionality. Sorting is done based on type hierarchy and popularity, whereas filtering is done to filter out APIs that aren’t public. However, these approaches require prior knowledge on the usage of each API which is unfeasible due to the drastic increase in the number of APIs in the past few years.

Bruch et al. [14] propose a Best Matching Neighbor (BMN) algorithm which is used to identify the nearest neighbors among the examples of API usages. These identified neighbors are then used for recommendations. This approach outperformed other techniques which
incorporated association-rule mining [15] and usage frequency.

Bayesian networks is another machine learning based approach that has been used to predict the next most likely method for code completion. Proksch et al. [30] use a Bayesian networks classifier along with context information to determine the likelihood of a method being invoked. These Pattern Based Bayesian Networks also incorporate clustering techniques to reduce the model size and increase efficiency. These Bayesian networks were more effective than BMN for the SWT framework in Java.

McCarey et al. [26] also analyze the effectiveness of using a Bayesian techniques for recommending library methods. However, the experiments show that a Vector Space Model outperforms the Naive Bayes, Bayesian Network, Tree Augmented Naive Bayes based classifiers.

Raychev et al. [31] model the extracted method call sequences into statistical language models like N-Gram and recurrent neural networks to predict recommendations. This approach has been proven to be fast and efficient in determining the likelihood of the next method call.

All of the above proposed improvements have been implemented for a static typed language, specifically Java. The main objective of these approaches is to filter out the less relevant methods or attributes for an object since they already have a prior knowledge on the static type and the methods that can be invoked on an object of that type.

Schäfer et al. [34] describe an approach based on static pointer analysis for smarter code completion results in JavaScript, a dynamic language. However, this analysis is flow-insensitive and thus, may not be able to detect some of the dynamic behavior noticed in Python like union types. Also, the APIs models used in this static pointer analysis method are generated after applying dynamic analysis on the framework’s test suite which may not be available for all frameworks or libraries for Python.
2.3 Evaluation of Recommenders

Most IDE tools depend on qualitative techniques and user studies to evaluate their effectiveness. However, it is very hard to find a representative set of users, and the approach could be time-consuming, costly and could result in subjective judgments [29, 16]. In this section, we will review research done in the area of automating this evaluation process to give a more objective idea of the performance of code completion systems.

Bruch et al. [16] propose an automated technique to evaluate the API recommenders [14, 14]. The technique is based on hold out validation which split the data into a training and test set. They use interpolated precision, recall and F1 measure as the evaluation metrics evaluate how their proposed recommendations fare. Though, these metrics are a useful indicator of the completeness and precision of the results, most of them are set-based measures and thus, aren’t very effective in describing the effectiveness of the ranked retrieval. Also, the relevant set generated to calculate these measures consists all the methods or attributes that a developer used on an object. Thus, it does not consider the order in which APIs are invoked in the evaluation.

Proksch et al [29] analyze the pros, cons and caveats in different approaches currently being used to evaluate the accuracy of the recommenders:

1. Manual case studies which consist of a few code completion scenarios that are identified and validated by experts

2. Automatic case studies in which completion scenarios are extracted from open source projects.

Among the approaches suggested, automatic case studies seems to be promising since the evaluation can be more comprehensive due to its ability to generate a large number of
scenarios as compared to the manual approach. However, the queries from the same project could cause over-fitting of the training model. Thus, in the cross-validation approach we plan to employ, such queries should be tested in the same fold to avoid the positive bias and strong correlation among the completion queries.
3

Approach

In PyReco, we extract object usages from several GitHub projects and use a nearest neighbor classifier on the extracted usage patterns to order our recommendations based on relevance.

The first phase of our implementation involves extracting the library object usages by applying static analysis on the abstract syntax trees of the python source files. Our AST Parser uses the abstract syntax trees generated by the ast [1] module in Python’s standard library. To order API proposals, we propose a Best Matching Object algorithm which is a Nearest Neighbor classifier based on Best Matching Neighbor (BMN) algorithm [14], to predict the next most likely method by using the mined usage information.

To illustrate our approach, we will use the code snippet described in Figure 3.1. The file handling example consists of two file objects, \texttt{fi} and \texttt{fo}, which are created using \texttt{open}. The example depicted shows a usage scenario wherein \texttt{fo}, a temporary file is created to dump the data present in \texttt{fi}. 
We used the advanced search API [11] of GitHub to extract open source projects rated with the most number of stars in Python. The number of stars in GitHub refers to the number of people watching the project.

Since APIs and software libraries are prone to change or become deprecated with time, the choice of such popular projects would be advantageous as they would be more likely to updated with API changes than a less starred one.

For our initial experiments, we extracted around 10,000 projects from GitHub [3].

### 3.2 Analysis of Python source files

In this part, we recursively walk through the nodes of the AST tree generated to analyze the python source files present in the GitHub projects. Some of the salient features of our analysis are as follows:

1. The python source code files are parsed in a top-down fashion. The top-down parsing
emulates the way forward-directed completion is done in an IDE.

(2) The library and module information is extracted from the Import nodes in the AST tree generated.

(3) An object assignment is added to the program’s graph if it has been created using a library function. The name of the identifier, argument types and values are stored as part of context information.

(4) An API method or attribute is recorded in the graph if the receiver object’s assignment has been previously recorded and the object is still alive in the current scope.

(5) The object’s death is marked when it is reassigned or when its scope ends.

We use graphs in our program analysis approach since it can be used to describe the assignments and calls in terms of the flow control of a program. The program’s graph splits when a branching or looping construct is encountered and merges on exiting that block. This splitting and merging of flows are depicted in the graph as shown in Figure 3.2. In the example shown, the graph splits on encountering the if-else block and the for loop in the program. These flows join after exiting the scope of these blocks.

To describe the control flow of the program, each node contains information on its entry and exit nodes as shown in Table 3.1. These entry and exit nodes help in the traversing across all the execution flows of the program and could be used to approximate the set of values an object can have at different points of the program using Reaching Definition Analysis[28] as shown in Table 3.1.

Reaching Definitions analysis is done to determine all the definitions that reach a particular point in the program. At a node $S$ in the graph, the reaching definitions $S$ is the union of the reaching definitions from the entry nodes minus the ones killed at $S$ (if an object dies at $S$) plus the definitions that are added in $S$ (if an object is reassigned at $S$).
For instance, at node 6 in the Figure 3.2, the reaching definition is a union of the definitions at 3, 4 and 5. No definition is added or subtracted at this node since the node at 6 marks a close call, not the death or assignment of an object.

These reaching definitions can be used to detect "union types". For instance, if fo was assigned to os.path(file) in the else block, the reaching definitions at 6 for object fo would be a set of values containing os.path and open. Thus, we could recommend methods that can be invoked on os.path as well as open.
Currently, our approach tracks assignments made using the assignment operator (‘=’) or using ‘with’ construct. An assignment node is added to the graph by evaluating the right side of the assignment expression for a library call.

Certain assignments such as that of the iterator in for loops is ignored since there is no substantial information on the type of the object. For instance, the object $s$ in the for loop shown in Figure 3.1 shows that it is a part of the iterator object $fi\.readlines()$ but the statement does not clearly indicate the type of $s$.

We are able to check “monkey patching” in most cases, by evaluating the left side of the assignment expression to check if a library method has been modified or overridden. Such calls are tracked and ignored while adding nodes to the program graph.

The program graph created after parsing the syntax trees is then used to extract call sequences for training the recommendation models. In the Figure 3.2, the extracted call sequence for $fo$ consists of a $close$ method, whereas for $fi$, the call sequence consists of $readlines$, $write$ and $close$ methods.

The assignment nodes in the graph also store additional context information like the initialization arguments. In the example depicted in the Figure 3.1, $file$ is stored for $fi$, and $temp$ and $w$ is stored for $fo$ as part of their context.

### 3.3 Best Matching Object Algorithm

After extracting the object usages from all the GitHub[3] projects, we train our models using a nearest neighbor classifier, which is based on the Best Matching Neighbor Algorithm [14]. We name this algorithm as Best Matching Object since it uses only the contextual information specific to the object to recognize the nearest neighbors.
Our tailoring in this approach are as follows:

(1) Vectors created for the training objects and query based on their invoking frequency.

(2) Manhattan distance is used as the distance criteria to select the nearest neighbors.

The basic approach involves creating vectors for the object usages extracted from the projects, computing the Manhattan distance with the frequency vector created for the query, and selecting the objects with the minimum distance from the query vector as the Nearest Neighbors. The recommendations are then presented based on the decreasing order of frequencies of methods invoked by the nearest neighbors.

The calls to different methods tend to follow patterns or chain sequences in a programming task. The object’s call sequence is always present in these frequency vectors generated since they indicate towards an object’s usage pattern. The call sequences for each object are recorded from the object’s assignment nodes to the nodes marking its death through a traversal of the program graphs generated.

Along with the method calls, the frequency vectors may also contain additional contextual information as shown in Figure 3.3.

A similar approach is followed while creating the frequency vector for the query during the evaluation process. A backward traversal of the graph is done to retrieve all the information (context and call sequences) related to the query object.
3.3.1 Additional context information

With the aim of extracting more contextual information on the object, we capture the following additional features from an object’s assignment:

1. Object’s name
2. The initialization argument values
3. The initialization argument types

Heinemann and Hummel [19] use the extracted identifier names in their nearest neighbor models to recommend methods. They argue that identifiers may contain important information on the intent of the developer. In cases wherein there are no prior methods calls or initialization arguments, we think that adding the identifier information would improve the recommendations. The processed identifier is stored in the training vectors after applying text processing and normalization techniques like Porter’s stemming algorithm [36].

The initialization arguments could be used as an indication towards the kind of object being created. In a dynamically typed language like Python, they can be indicators towards the type of the object being returned by the API. For instance, in a file handling example, the

```python
file = "samples/sample.txt"
temp = "samples/temp.txt"
fi=open(file)
fo=open(temp, "w")
for s in fi.readlines():
    if s.strip():
        fo.write(s)
    else:
        fo.write("some string")
fo.close()
fi.close()
```

Figure 3.3: Encoding the file objects as frequency vectors

```markdown
| fi | 0 1 0 1 0 1 0 0 1 |
| fo | 0 1 0 2 1 1 1 0 |
```
optional flag could be an indication of an object usage. For instance, a file object created with the ‘w’ flag indicates more towards write operations.

### 3.3.2 Selection of Nearest Neighbors

To identify the nearest neighbors for the completion query, we compute the Manhattan distance between the query vector and other similarly defined objects found in our training dataset. The distance measure is calculated by taking the sum of the absolute values of the
differences between the variables of the two vectors as shown in the following formula:

\[ d = \sum_{i=1}^{n} |x_i - y_i| \]

Manhattan Distance [21] was selected as the distance measure for measuring the similarity between the feature vectors since it performed much better than other measures like Euclidean distance in our initial experiments.

Unlike the Best Matching Algorithm described by Bruch et al.[14], we kept the original frequency values instead of reducing them to boolean factors since some methods tend to be called more frequently than others. To calculate the Manhattan distance we considered only the number of variables present in the query vector.

In Figure 3.4, open obj1 and open obj2 are selected as the nearest neighbors since they have the minimum Manhattan distance among the extracted open objects. The nearest neighbors then vote based on their method frequencies. This vote is done subtracting the method frequencies already present in the query vector. The methods that are recommended in the example query shown in the Figure 3.5 are readline, read and close.
Evaluation

In this section, we will describe the automated process, metrics and the dataset that were used for measuring the effectiveness of the code recommenders.

4.1 Experiment Procedure

To evaluate the effectiveness of PyReco in giving recommendations, we implemented a 10-fold cross-validation as described in Figure 4.1 which consists of the following main features:

(1) In each fold, 10% of the library objects are kept aside as the validation set while the other 90% are used for training.

(2) The validation set varies with each fold of the cross-validation and queries for each fold are selected randomly. Since our initial experiments and the experiments described in Section 2.3 showed that queries from the same project result in over-fitting the model, the intra-project usages are assigned to the same fold.

(3) We query the recommenders for every method call made by these library objects by
calling their code completion API, thus making the process completely automated.

(4) After each query, the evaluation metrics are calculated with the relevant set containing
the method which the user had originally used in the program. Thus, the relevant set
in our approach contains only one method.

4.2 Experiment Dataset

In our experiments, we tested the recommenders for the objects created using re [2], a Python
standard library and a third party library, django [12]. Django is a popular open source web
framework whereas re is a module used by developers for dealing with regular expressions.

These libraries were chosen due to their popularity and due to the large number of objects
found in our dataset of software projects.

4.3 Evaluation Metrics

To measure the quality of predictions made by the code recommenders, we used Mean Reciprocal Rank
(MRR), Recall and 1 – Precision.
Mean Reciprocal Rank is calculated by averaging the reciprocal of the rank at which the first relevant document was found across all the information needs [18]. For a set of code completion queries $Q$, the Mean Reciprocal Rank is defined as:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}$$

Since this measure captures the rank of the relevant result, it does not penalize the systems which retrieve long lists as Precision does. In our case, this measure becomes equivalent to Mean Average Precision (MAP) since our relevant set contains only one element.

We use Recall to estimate the “completeness” of our results. It gives us an idea of the number of times the recommendation list did contain the method which the user was looking for. Recall [28] is calculated using the following formula:

$$\text{Recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

R – Precision is another measure which is commonly used for ranked retrieval. For a set of known relevant results of size $R$, this value is defined as the precision of the top $R$ documents retrieved [28]. We calculate the precision at $R = 1$ in the evaluations since the relevant set contains only one result. This measure gives us an estimate of the number of times the relevant result appeared at the top of the recommendations.

Precision at $k$ is another metric in ranked retrieval which computes the Precision for the top $k$ retrieved results. R – Precision was preferred when compared with Precision at $k$ since it is much more stable when the number of recommendations retrieved is less than $k$ [28]. However, this value tends to be a bit harsh when the value of $R$ is 1 since it only considers the top most result among the suggested completions.
Among the three measures described, we consider $MRR$ as the main criteria to decide the quality of the results when the metrics have contrasting values.

To illustrate the way these metrics are calculated, consider the code recommendations (``read-line, read, close``) made in Figure 3.5. If `read` was the method the user used for the query shown in Figure 3.4, then the values of the metrics will be as follows:

1. $MRR = 0.5$ since the rank at which `read` was found is 2.
2. The Recall for this query is 1.0 since `read` was in the set of retrieved recommendations.
3. The value for $1 - Precision$ is 0 since `read` was not found at the top of the recommendations.

### 4.4 JEDI

JEDI [4] is a static analysis tool for Python which has been integrated into IDEs such as Atom. It has also been provided as a plugin for text editors such as Sublime. JEDI provides an API which can be used for retrieving the code completion results which is ideal for the automated evaluation approach described in Section 4.1. The results returned by this API provided by JEDI are alphabetically ordered.
5

Results and Findings

In this section, we will discuss the results and findings of our experiments conducted to assess the effectiveness of PyReco as a code recommender. To evaluate the performance, we extracted a total of 31,581 completion queries for *django* and 31,480 queries for *re* which were sent across ten different splits for cross-validation.

5.1 Impact of Additional Context Information

As described in Section 3.3.1, we considered several additional contextual features to improve PyReco’s ability to predict the next method call more accurately, especially in guessing the first method call that can be invoked on an object.

To evaluate the impact of this addition of context information, we incorporated a technique similar to forward selection in Feature Selection, to analyze which context feature(s) capture the most information on the usage of a library object. This was done by plugging in these features incrementally in the model, and using the computed MRR and Recall values to judge the quality of the results.
There were three kinds of additional context information – initialization argument values (AV), initialization argument types (AT), object’s name (ON) – for which feature selection was done. We also added the base case (None) which only uses the information on the prior call sequence to order the recommendations. Figure 5.1 and Figure 5.2 summarize the results of our experiments on *django* and *re* for this analysis.

In terms of MRR, we notice that the model containing the object name (ON) performs best for *django* whereas a combination of object name, initialization argument value and type (ON, AV, AT) performs best for *re*. The MRR values for these models are greater than the base model which only contains information on the methods that were previously invoked.
Similar trends were noticed in the results for Recall. The model with the best MRR value also had a high value for Recall. Though the addition of some of these contextual features improves upon the base case of PyReco for both re and django, this observed improvement is almost negligible in both Precision and Recall. The MRR values is boosted by 1% in re whereas in django, the increase in precision is even lesser (0.3%).

Contrary to our expectations, we notice that the addition of contextual features does not significantly impact the predicting capability of our recommender. In the following experiments, we will consider the base model without the additional features for further comparisons.

5.2 Prediction with different types of queries

In this section, we evaluate the influence of the prior call sequence in predicting the results accurately. We expect the order of invocation to be an important factor that influences the results of our recommendation model since it can capture a user’s goal in using the API. Thus, we believe that the more a call sequence in an API’s usage pattern is known, the more accurate the recommendations will be. To analyze this hypothesis, we evaluate PyReco with queries which differ based on their order of invocation. The results of this experiment are depicted in Figure 5.3.

The experiment shows the results for re does behave as our initial expectations. The MRR value for re is boosted from a MRR value of 0.715 to 0.779. However for django, the MRR value drops to 0.297 from a value of 0.521.

To investigate this contrasting behavior, we recorded the number of object usages found for the different query groups. The results for this evaluation are shown in Figure 5.4. However, the number of object usages could not be used to explain the behavior for re since there were more usages found for django that found for re.
One reason that could explain the results for django is that the number of usages found in the dataset is not complete. This explanation is corroborated with recall values shown in Figure 5.2. The recall value for re is around 99% whereas the value for django is 65%. This indicates that the relevant results were not in the recommendations for 35% of the cases. To improve the results for django, we could increase our training dataset.

To evaluate the influence of the number of usages for django, we increased our dataset of projects to 20,000. With the addition of 5,000 new object usages to our training models, we compared the prediction quality of the completion results as shown in Figure 5.5 and
Figure 5.5: Impact of number of object usages on evaluation metrics

Figure 5.6: Impact of number of object usages on query types

Figure 5.6. The increase did impact the evaluation metrics as predicted. We noticed a 3% and 3.5% increase in MRR and recall values, respectively. This indicates that the rise in training objects did improve the recommendations made by PyReco.

The comparison after increasing the training set also shows an improvement for the different query types. We observed that the method-call prediction for the second and third order of invocation increased with this rise in objects found in software projects.

Thus from the experimental data, we can infer that the recommendations for a call sequence becomes more accurate only if the dataset is complete with the API usage patterns. The high MRR values for re does signify the completeness in the usage patterns found in our
projects. We would need to train our models over a greater set of projects to improve the relevance of the code recommendations for *django*.

### 5.3 Comparison with JEDI

In order to compare the performance of PyReco with JEDI, we sent the same set of completion queries for *re* and *django* using the 10 fold cross validation technique described in Section 4.1. The evaluation metrics were averaged across the queries and are listed out in Figure 5.7 and Figure 5.8.
The results for re and django show that PyReco does outperform JEDI in terms of prediction quality and completeness of recommendations by a huge margin.

PyReco’s MRR value for re is 0.721 which implies that on an average, the relevant result is found between the first and second position in the recommendation list. On the other hand, for django, the MRR value is 0.417 which implies that on an average, the relevant result lies between the second or third position.

The recall measures also indicate the effectiveness of PyReco’s results in predicting the method in the queries provided. Though the number of results is limited to 10, the relevant result is discovered 99% of the time in re and 65% of the time in django.

The 1-Precision results give an idea of whether the method suggested at the top of the recommendation list matches with the user’s choice. The desired method was at the first position for 57% of the cases for django and 30% for re in the completion suggestions.

These experiments also point towards the ineffectiveness of JEDI to recommend methods for django objects. JEDI fails to propose auto-complete suggestions for the code completion queries for django in most cases.
6

Conclusion

In this thesis, we described the working of PyReco, a code recommender, to help software developers explore APIs of libraries and frameworks more effectively and efficiently in Python.

The proposed system addresses the challenge of recommending APIs in a dynamic language by reusing the intelligence found in open source repositories on API usages and suggests method calls that are ordered by relevance, unlike the other code completion tools currently available for Python.

Our experimental results show that the predictions made by PyReco are much more precise and complete as compared to JEDI [4] for the standard and third party libraries tested.

6.1 Threats to Validity

We identified the following threats in our described approach:
(1) **Generalization based on tested libraries**

Our results are summarized based on our experiments using two library methods, `re` and `django`. However, these findings could be challenged when other libraries are used. Our choice of the libraries was based on the high frequency of usage patterns detected in our dataset of projects and its popularity in the Python development community.

(2) **Presence of bugs in repositories**

There is a possibility that some of the source code files used for training may contain bugs and thus may lead to some false positives in the recommendations. However, since the projects extracted are the top starred ones, the presence of these bugs is expected to be negligible.

(3) **Comparison with JEDI**

PyCharm [8], IntelliJ’s plugin for Python, has a much more popular and powerful code completion feature than JEDI; however, their implementation is closed and hence cannot be used for our experiments.

### 6.2 Future Work

We plan to integrate PyReco as a plugin in the current Integrated Development Tools (IDEs) for Python. However, in order to do that, we need to conduct user studies with developers to assess the usefulness of such a plugin for directions in its design.

The prediction of relevant method-calls for a library in our tool depends on PyReco’s ability to capture all possible usage patterns from the repositories. In our experiments, we found that extracting patterns from 10,000 projects were insufficient for `django` whereas the number sufficed for `re`. Thus, we need to scale up the number of projects for extracting the usage patterns for libraries with large usage scenarios like `django`. 

34
We could use manual case studies [29] to evaluate the effectiveness of PyReco with smarter code completion tools like PyCharm [8]. The results of such an experiment would give us a better idea of the impact of a relevance-based ordering of completion results especially in the case of *django*.

Another future direction in our project could be implementing our own parser to generate the Abstract Syntax Trees for the static analysis part. Our current approach requires parsable code to generate AST trees since it relies on the *ast* [1] module provided in Python’s standard library. However, code completion snippets are incomplete and not always parsable. Although we currently do have a basic feature in PyReco to handle some of these scenarios, an inbuilt parser would be more useful to solve issues that were not foreseen during our experiments.
Bibliography


