PLIR: A Research and Experimentation Platform for Natural Language Code Search

Will Hawkins
Department of Computer Science
University of Virginia
whh8b@virginia.edu

ABSTRACT

There exist myriad tools that allow programmers to search source code. Some of these tools are as old as UNIX (e.g., grep) and some are newer than Google. Some run on a local computer and some run over the Internet. However, few tools allow the user to search a code repository using natural language queries. Because of this restriction, most of these tools are not an effective means for developers to mine the massive amount of publicly available source code for idiomatic code snippets, the best implementations of a particularly complicated algorithm or the most well-tested submodules or functions to accomplish a task. PLIR is an extensible system for developing, researching and deploying natural language source code information retrieval systems. It is built on techniques and systems from the fields of programming language design and analysis and information retrieval. The goal of PLIR is to combine existing tools in a way that makes it easy for developers and researchers to study and enhance each of the subcomponents that comprise a functional natural language source code information retrieval system. This paper will describe the implementation of PLIR and show how it can be used to evaluate hypothesis about improvement on existing information retrieval techniques related to natural language processing of computer source code.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing – *Indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – *Query formulation, retrieval models, search process, information filtering*; D.2.6 [**Software**]: Programming Environments – *Programmer workbench*; D.2.13 [**Software**]: Reusable Software – *Reuse models*; D.3.2 [**Software**]: Language Classifications – *Very high-level languages*;

General Terms

Languages, documentation, experimentation.

Keywords

www, source code search, source code categorization, search engine

1. INTRODUCTION

There are plenty of tools that allow programmers to search source code. Some of these tools are as old as UNIX (i.e., grep) and some are newer than Google. Some run on a local computer and some run over the Internet. But none of them allow the user to search a code repository using natural languages queries. This is a major restriction that keeps existing code search tools from being used to search the wealth of openly published source code for idiomatic code, the best implementations of a particularly complicated algorithm or the most well tested components to accomplish a particular task.

The ultimate “programmer's assistant” is a tool whose input is the natural language description of an algorithmic task and whose output is the source code to accomplish that task. In order to build a system to achieve that goal, at least four major subsystems are needed. First, there must exist a system to correlate source code with documentation and programmer comments in order to generate natural language descriptions of existing blocks of code. Second, there must exist a formal system that automatically generates inputs, outputs and pre- and post-conditions of that source code. Third, there must exist a system that takes natural language algorithmic task descriptions and retrieves a series of snippets that, when properly integrated, accomplish that task. Fourth, and finally, there must exist a system that properly composes those snippets by handling differences in expected inputs, expected outputs, etc.

Even when examined at a very high-level, building such a programmer’s assistant seems like a very difficult task. In fact, building any one of those subsystems would be a major accomplishment.

This paper describes PLIR, an extensible system for developing, researching and implementing natural language source code information retrieval systems. It is built on techniques and systems taken from the fields of programming language design and analysis and information retrieval. The goal of PLIR is to combine existing tools in ways that make it easy for developers and researchers to deploy, enhance and evaluate each of the components of a functional natural language source code information retrieval system.

This paper will describe the implementation of PLIR and show how it can be used to evaluate hypotheses for improving state-of-the-art information retrieval techniques for natural language processing of computer source code. The hope is that by using PLIR, researchers can study and improve the individual components necessary to create the Utopian programmer's assistant.

PLIR is built like any modern information retrieval system. It includes a crawler, analyzer, data store, indexer, analysis engine, document retriever and ranker. Each of these components is decoupled from the others. Components communicate using well-documented APIs. At the core of PLIR is the Documentation Database (DocDB). Crawlers download source code and feed it to analyzers. General purpose or language-specific analyzers digest the crawler's output and store the results in the DocDB. Indexers and analysis engines[1] (AZR) operate on the data in the DocDB before it is imported into document retrievers and rankers. Once the source code has made its way through the pipeline, end-users can retrieve that code using natural language queries.

[1] The analysis engine component is hereafter referred to as AZR to avoid confusion with the analyzer component.

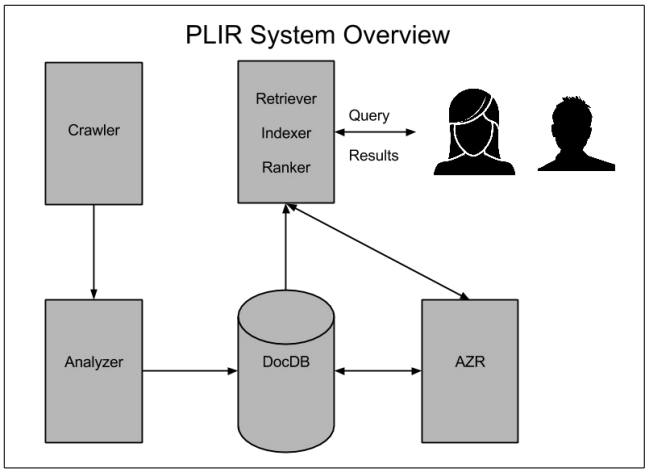

Figure 1: PLIR system overview.

1.1 Related Work

Google operated its own Internet-scale source code search tool, known as Code Search, between 2006 and 2011. [2] It was a powerful search tool and allowed the user to input regular expressions to search a large corpus of source code. Even years after it shut down, Code Search is the only Internet-accessible code search tool to ever let users input regular expression queries.

Searchcode is another Internet-accessible code search tool. Searchcode allows the user to "[s]earch over projects from Github, Bitbucket, Google Code, Codeplex, Sourceforge, Fedora Project and more." [16] The user specifies their information need using a specialized query that allows them to limit results based on programming language, data type, etc.

Krugle Opensearch is an Internet-accessible code search tool based upon the company's proprietary enterprise code analysis suite. It "is a free, online version of Krugle Enterprise V5 that can be used to test drive the capabilities of our Enterprise solution - using popular open source software projects including Apache and Mozilla Firefox." [5]

Finally, consider a tool known as Codase. Codase is billed as an advanced source code search engine *for the very reason that it lets the developer query in pseudo code*. "For example, to find any main method that contains variable t and function calls of thread.start() and println, this query can be used: main() { var t; thread.start(); println; }." [7] Although that type of query may be useful for someone that already generally knows what they want to find, it does not suit the needs, for example, of a developer who does not even know how to generally describe the outline of a solution for a task outside their area of expertise.

There are also source code search tools that run on a user's local computer or Intranet. These tools index local code repositories and make their results available only to internal users.

OpenGrok is such a tool. "OpenGrok is a fast and usable source code search and cross reference engine. It helps you search, cross-reference and navigate your source tree." [12] Although it is built for local deployment, the developer's advocate its use for those who want to share reusable open source code: "It lets people easily and quickly find the source code, look at it, understand the history and changes made to the source. It makes a developers [sic] life easy. It certainly has made my life easy when I am looking for security holes in software." [20]

While each of these tools may have certain features that make them useful, none of them allows the user to input natural language queries to retrieve relevant snippets of code.

Beyond industry, there is academic research on this topic. Sridhara et. al have done work on discovering the relationship of words in source code [17], capturing the context of source code to improve the performance of natural language query results [4] and building natural language descriptions of program functionality from its source code. [18] Masaru Ohba et al have proposed a "novel technique to efficiency [sic] mine *concept keywords* from identifiers in software projects." [11] Andrian Marcus et al have attempted to use latent semantic indexing to "map concepts expressed in natural language by the programmer to the relevant parts of the source code" [9] and Scott Grant and James R. Cordy examined the number of unique concepts required to represent the functionality of any particular "substructure [of] a source code corpus." [15]

The utility of each of the systems generated in these works are tested by the individual authors on only a small set of source code. For instance, in [9], the authors test their approach on the source code of only a single program. The data collected by a PLIR crawler will provide a corpus to test and validate ongoing academic research in natural language code search.

1.2 Outline

Section 2 describes the implementation PLIR itself and describes the implementation of two language-specific analyzers. Section 3 evaluates an information retrieval system for natural language processing of program source code using a basic PLIR instantiation. Section 4 gives an example of an analysis engine built using the PLIR framework and evaluates its performance relative to the results presented in Section 3. Section 5 lists several areas of future work.

2. IMPLEMENTATION

2.1 Documentation Database

The Documentation Database (DocDB) is the heart of the storage and retrieval mechanism of PLIR. It holds all the package identifications, documentation and source code as well as the dependencies and connections therein. Packages, documentation, source and dependencies are stored in separate tables and normalized with unique identifiers.

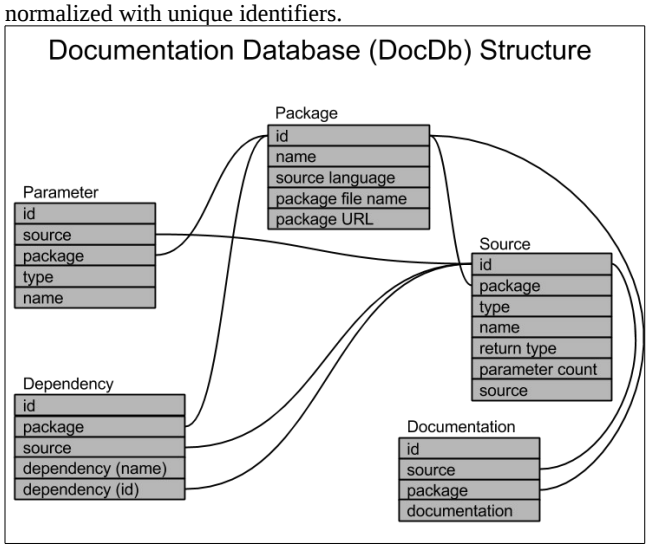

Figure 2: Documentation Database (DocDb) structure.

2.1.1 DocDb Structure

The Package table stores information about software packages. Each package has four associated fields of metadata:

1. The name of the package
2. The programming language used by the project
3. The filename of the package source code
4. The URL of the package

(1) is the name of the package as determined by the crawler. The crawler determines the name of the package depending on how the code is retrieved. Field (2), the programming language, is also determined by the crawler. Fields (3) and (4) are related to one another: They identify the original location of the code for later presentation to the system user.

The Source table stores information about source code for packages. The DocDB considers three different types of source code: class, namespace and method. While only the identifiers for classes and namespaces are kept in the DocDB, the entire source code of methods are stored. For this reason, the design of the source table is tailored to methods:

1. A link to the package
2. The type of the source
3. The name of the object
4. The return type
5. The parameter count
6. The source code itself

Field (1) is a pointer to the package that contains this namespace, class or method. Field (2) is the type of the source code: class, namespace or method. Field (3) is the namespace, class or method name. Fields (4), (5) and (6) are meaningful only for methods. Field (4) is the method return type. Field (5) is the number of method parameters. Although the types and names of parameters are stored separately, the parameter count is useful to differentiate polymorphic functions with variable number of parameters. For more information, see Section 5. Field (6) is the source code of the method itself.

Even though the source code for methods and namespaces are not kept, it is important to store their identifiers in the DocDb for two reasons. First, documentation formats like Doxygen and JavaDoc allow for specific comments at these levels of abstraction. Second, the identifiers themselves are used as dependency links. More on dependency links below.

The Parameter table stores information about the parameters of a particular method. It has four fields:

1. A link to the method
2. A link to the package
3. The type of the parameter
4. The name of the parameter

Field (1) is a pointer to the row in the source table that holds the source code for the method that uses this parameter. Because a function may have multiple parameters, links between the parameter table and source table are many-to-one. Field (2) is a link to the package. This could be derived through Field (1) but holding it here makes later retrieval easier. Field (3) is the type of the parameter and Field (4) is the parameter name.

The Dependency table stores information about dependencies between packages. Dependency analysis is performed with respect to methods and each method is determined to depend on zero or more packages or namespaces. The calculation of dependencies is described below. To store the dependencies, the Dependency table has four fields:

1. A link to the method
2. A link to the package
3. The name of the dependency
4. A link to the dependency

Field (1) is a pointer to the row in the source table that represents the function that has this dependency. Field (2) is a pointer to the package that contains this method. As in the Parameter table, this could be derived through Field (1) but holding it here makes later retrieval easier. Either Field (3) or Field (4) are defined, but not both. Field (3) is the name of a dependency, e.g. `com.android.View`. Field (4) is a pointer to the row in the source table that represents the dependency. In other words, if the crawler has analyzed the source code of the `com.android.View` package, all the entries in this table with a dependency on that name would be replaced with a pointer to the appropriate row in the source table.

The Documentation table stores documentation associated with a source object. The Documentation table has four fields:

1. A link to the source
2. A link to the package
3. The documentation text

Field (1) is a pointer to the row in the source table whose object is being documented. The object may be a class, namepsace or method. Field (2) is a pointer to the package that contains this method. As in the Parameter table, this could be derived through Field (1) but holding it here makes later retrieval easier. Finally, Field (3) contains the text of the documentation itself.

2.1.2 API

To facilitate its use by researchers and other PLIR deployers, there are libraries that facilitate interaction with the DocDB. Libraries exist for C++, Python, PERL and Java. These libraries abstract database-specific concepts so that researchers can focus on adding information to the DocDB and/or analyzing existing content.

2.2 Crawler

Given the normalized, language-agnostic design of the DocDB, there are many types of programming projects that can be stored and analyzed. Most existing research on natural language analysis of source code has focused on projects available at Sourceforge, Freshmeat, the Apache Foundation, etc. [8],[18] The fields in the DocDB have been designed to handle the most common types of data associated with projects on those sites.

In terms of behavior, there are no constraints on the crawler. The implementor may decide on depth-first, focused crawling of a website that hosts code or may implement a crawler that searches through Google results. See Section 3.1 for a description of an exemplary crawler that downloads source code hosted on GitHub.

The Crawler hands off information about code that it downloads to the Analyzer.

2.3 Analyzer

Source code analysis is performed by Analyzers. Analyzers are source language-specific and adhere to a Runner API. This API is

implemented in Python. For every repository given to the Analyzer by the Crawler, a language-specific Runner is invoked. The runner is executed on that repository. The runner is responsible for doing language-appropriate analysis of the code in the repository and properly filling the DocDb.

Because the Runner is given complete independence over analysing the source code and filling the DocDB, Runners do not need to be written entirely in Python. In fact, the two Runners in existence are written in Java and C++ (to analyze projects written in those languages). The Runner API is simply a way of passing information about a repository from the Crawler to an instantiation of the Analyzer.

2.3.1 The Java Runner

The Java Runner is implemented in Java and uses the standard Doclet and JavaCompiler API. Analysis of projects in Java using the Java Runner proceeds in two steps: First, JavaDocker stores source code documentation in the DocDB. Second, JavaSource stores the source of the project in the DocDB. In the process, if there is documentation associated with that source code, the two are linked. The Python code that implements the Runner API simply calls out to the JavaDocker and JavaSource executables.

JavaDocker implements the Doclet API, and gets programmatic access to all comments in the source code that adhere to the JavaDoc syntax. This means that comments available to the JavaDocker are associated with a particular package, class or method. The JavaDocker inserts the documentation into the Documentation table of the DocDB and associates it with a source "stub" that contains the source name (and perhaps return type and parameters, in the case of a method).

JavaSource implements several APIs that allow the tool to programmatically invoke the Java compiler and traverse the resulting abstract syntax tree. As each Java file is parsed by JavaSource, the classes and methods are extracted. Each of these is inserted into the database by either updating an existing entry or adding a new row. Existing entries are updated when the class or method was previously inserted by JavaDocker. Entries are inserted when the class or method has never been seen before (which implies that it is not documented according to JavaDoc standards).

To calculate dependencies, JavaSource pays attention to `import` statements. By convention, in Java `import` statements affect an entire .java file; packages are imported at the top of the file and those packages are used throughout. For each .java file, JavaSource compiles these imports into a list. Then, for each method seen in that source file, those packages are considered dependencies. Dependencies are added into the Dependency table of the DocDB by name when the dependent namespace does not already exist. If the dependency already exists in the DocDB, the ID for that source code is used as the link instead of the name. While it is certainly not the case that each method depends on each one of those packages, it is is still a meaningful approximation.

In the course of traversing the AST, JavaSource also looks for package declarations. These declarations are the satisfying end of a dependency relationship. When a package is declared, the dependency table is updated to reflect the fact that code satisfying the dependency has been analyzed and now exists in the DocDB.

2.3.2 The Cpp Runner

The analyzer for projects written in C++ is implemented in PERL (using a Doxygen API) and C++ (using Clang). As in the case of the Java Runner, analysis of projects written in C++ using the C++ Runner proceeds in two steps: First, CppDocker stores source code documentation in the DocDB. Second, CppSource stores the source of the project in the DocDB. In the process, if there is documentation associated with that source code, the two are linked. The Python code that implements the Runner API simply calls out to the CppDocker and CppSource executables.

CppDocker invokes Doxygen on the entire set of files in the project. Doxygen considers only those files that are code (headers or implementation) and skips everything else. When invoked, Doxygen is configured to output its documentation in a special PERL module format. CppDocker includes these modules to programmatically iterate through generated documentation for methods, classes, namespaces, etc. Like JavaDocker, when CppDocker inserts documentation for source code, it also creates and inserts a “stub” source object. Depending on what is available from the output of Doxygen, these stubs have information about parameters, return types, etc.

CppSource is written as a Clang front end action. Front end actions are tools that can use the output from the Clang compiler programatically. CppSource traverses the abstract syntax tree for each of the source files and finds class and method declarations. Entries for these declarations are either updated or inserted in to the Source table of the DocDb. Existing entries are updated when the class or method was previously inserted by CppDocker. Entries are inserted when the class or method has never been seen before (which implies that it is not documented according to Doxygen standards).

To calculate dependencies, CppSource pays attention to `using` namespace statements. C++ `using` namespace statements typically affect an entire scope and programmers normally invoke them at the top of source files. For each C++ file (implementation or header), CppSource compiles these namespaces into a list. Then, for each method seen in that source file, those namespaces are considered dependencies. Dependencies are added into the Dependency table of the DocDB by name when the dependent namespace does not already exist. If the dependency already exists in the DocDB, the ID for that source code is used as the link instead of the name. Again, although it is certainly not the case that each method depends on each one of those namespaces, it is is still a meaningful approximation.

In the course of traversing the AST, CppSource also looks for namespace declarations. These declarations are the satisfying end of a dependency relationship. When a namespace is declared, the dependency table is updated to reflect the fact that code satisfying the dependency has been analyzed and now exists in the DocDB.

2.4 Analysis Engine - AZR

Each row in the Source table of the DocDB holds information about a particular class, namespace or method. Linked to that row is a) the source code (if the row represents a method), b) a set of entries for the type and name of each parameter (again, if the row represents a method) and c) a set of pointers to and from dependencies on or of that row. Thanks to the structure of the DocDB and the information stored therein, a modular set of AZRs can be implemented to influence retrieval performance. Depending on the type and function of an AZR, it may store its results in the DocDB (where they are later incorporated in the indexer, retriever and ranker) or it may store its results in another location for online access when the indexer, retriever and ranker are responding to a user's query. See Section 4 for one an implementation of an exemplary AZR and its effect on performance.

2.5 Indexer, retriever and ranker

Indexing, retrieving and ranking in PLIR are all done using Solr. "Solr is a standalone enterprise search server with a REST-like API" [15] that "uses the Lucene search library and extends it." [15] Solr allows the PLIR implementer to customize the indexing, retrieving and ranking of the information stored in the DocDB and is the basis for end-user interaction.

Solr handles PLIR queries, parses them and generates a set of results from the DocDB. PLIR comes with a default configuration that allows the end user to search documentation only or code only. In essence, the PLIR configuration instructs Solr to treat documentation and source code as separate documents. Solr can dynamically combine these two options so that the user's query searches a combination of the two. This is just one way that Solr can be configured to operate in the PLIR system.

Solr compares the end-user's query with documents (i.e., source code and documentation) in its index. Solr can be configured to tokenize documents in myriad ways. This means that the PLIR implementer has the ability to tune the way that result candidate documents are generated depending on the source language. For example, if the document is Java source code, the PLIR implementer might tokenize the source code and split variables in accordance with Camel Case naming conventions. One concrete implementation of such a tokenizer will be described below.

The candidate results are ranked according to Solr's configuration and the PLIR implementer can choose from several similarity metrics (TF/IDF, BM25, Jelinek-Mercer, Dirichlet). Again, the PLIR implementer has the opportunity to tune the ranking according to his/her specifications.

3. RESULTS

To understand baseline performance of a PLIR instantiation, researchers deployed a PLIR instance using popular C++ and Java repositories on GitHub. This PLIR instantiation is evaluated in the context of its role as a component in a programmer's assistant.

As described in Section 1, the goal of a programmer's assistant is to “generate” source code from natural language. As such, documentation linked with source code is a helpful hint for the system, but does not affect the system's utility to the user *per se*. This distinction will become evident below with the definition of the evaluation metric.

3.1 Crawler

This particularly crawler works by feeding the Analyzers popular repositories from GitHub. GitHub refers to these as trending repositories and categorizes them by language. In other words, it is possible to search for the trending Javascript repositories or the trending PHP repositories.

The crawler takes as input the name of a language: Javascript, Java, C++, C, etc. It queries GitHub for the names of the popular repositories whose source code language matches the user input. Iteratively, the crawler clones each popular repository and passes control to the analyzer.

3.2 Analyzer

Depending on the language of the repository, the Java Runner or the Cpp Runner take care of analyzing the source code and storing the results in the DocDb. The basic PLIR instantiation used for this experiment contained more than 81,000 comment blocks, 380,000 methods and more than 140,000 dependencies across 66 different packages.

3.3 Indexer, retriever, ranker

For this experiment, researchers fixed a tokenizing configuration for Solr and varied the configuration of similarity ranking functions to judge performance.

For code documents, researchers configured the Solr tokenizer to split source code tokens by Camel Case changes and _s. In other words, in this experiment, Solr split FunctionCall into Function Call and parameter_one into parameter one. For documentation documents, researchers configured Solr to ignore the Solr-standard set of stopwords. For both document types, case was ignored.

The user's query was matched against both the source code and the documentation available in the DocDB. Matches in either were evenly weighted.

3.4 Evaluation

One of the packages downloaded by the crawler, OkHttp, was chosen for experimentation because of the availability of external documentation. OkHttp is a Java library for performing SPDY HTTP requests and has a cookbook of recipes that describe canonical library usage. [13]

For the test, researchers used the recipes for synchronous and asynchronous HTTP file transfer. Each recipe describes the task (in prose) and gives an accompanying snippet of code to accomplish it.

Task 1: Synchronous HTTP file transfer using OkHttp

Task 2: Asynchronous HTTP file transfer using OkHttp

Researchers defined Golden Code (C_g), library-specific function calls and variables, for the code that accomplishes Tasks 1 and 2. Researchers identified 16 and 19 OkHttp-unique components in the synchronous and asynchronous recipes, respectively. See Text 1 and Text 2.

```
private final OkHttpClient client = new OkHttpClient();

public void run() throws Exception {
    Request request = new Request.Builder()
        .url("http://publicobject.com/helloworld.txt")
        .build();
    Response response = client.newCall(request).execute();
    if (!response.isSuccessful()) throw new
        IOException("Unexpected code " + response);

    Headers responseHeaders = response.headers();
    for (int i = 0; i < responseHeaders.size(); i++) {
        System.out.println(responseHeaders.name(i)
            + ": " + responseHeaders.value(i));
    }

    System.out.println(response.body().string());
}
```

Text 1: Source code (with Golden Code bolded) for Task 1.

```
private final OkHttpClient client = new OkHttpClient();

public void run() throws Exception {
    Request request = new Request.Builder()
        .url("http://publicobject.com/helloworld.txt")
        .build();

    client.newCall(request).enqueue(new Callback() {
    @Override public void onFailure(Request request, Throwable throwable)
    {
        throwable.printStackTrace();
    }

    @Override public void onResponse(Response response) throws IOException
    {
        if (!response.isSuccessful())
            throw new IOException("Unexpected code " + response);

        Headers responseHeaders = response.headers();
        for (int i = 0; i < responseHeaders.size(); i++) {
            System.out.println(responseHeaders.name(i) + ": " +
                responseHeaders.value(i));
        }
        System.out.println(response.body().string());
    }
    });
}
```

Text 2: Source code (with Golden Code bolded) for Task 2.

In the experiment, researchers played the role of the external developer who wants to use OkHttp in his/her project. Researchers used the recipe's descriptive text to generate queries in accordance with this usage scenario.

Task 1 Query: Download a file, print its headers, and print its response body as a string.

Task 2 Query: Download a file on a worker thread, and get called back when the response is readable.

To evaluate the quality of results, researchers scored the code of each PLIR result using a Code Relevance (CR) metric.

$$CR = \sum_{c \in C_g \wedge c \in C_r} 1$$

Equation 3: Formula to calculate CR for a result.

In Equation 3 C_g is the Golden Code set and C_r is the code from the result being scored. The CR metric is like a relevance value in a traditional IR system in that it considers relevance to be more than a binary judgment. The score accounts for the presence or absence of the previously identified OkHttp-unique function calls and variables required to accomplish the task. For instance, if a developer must instantiate a `Request.Builder` to download a file synchronously, then the evaluation metric for a result would increase by 1 if it included, for example, `Builder b = new Request.Builder();`. Additional points are not given for repeated use.

In each test, the first ten PLIR results were scored and summed to calculate a cumulative CR (CCR@10). The CCR@10 does not consider the relative position of result within the set of the top ten results. In this way, CCR is like Precision@K. Nevertheless, as shown in Section 4.1.2, changes to the quantitative CCR score do reflect subjective changes in system performance.

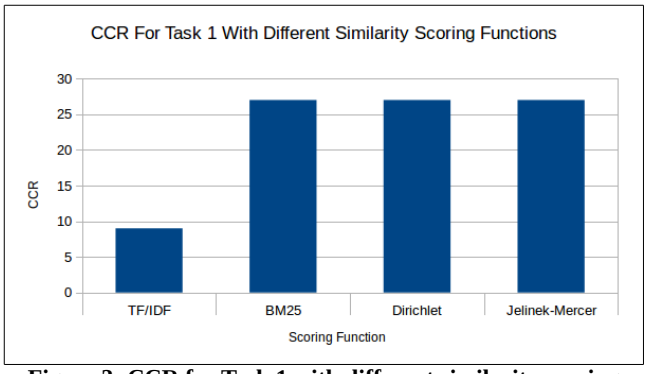

Figure 3: CCR for Task 1 with different similarity scoring functions.

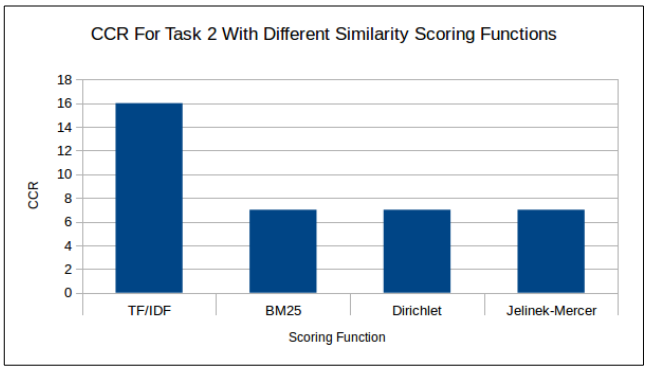

Figure 4: CCR for Task 2 with different similarity scoring functions

Generally, this PLIR instantiation performed better on Task 1 than Task 2. This performance gap is the result of word choice for the queries. The Task 1 Query contains words and phrases that are not only present in the documentation but are also present (identically) in the source code. Remember that the CCR considers only the source code of the results and weights tokens from the result's source code and documentation evenly. Therefore, while Task 2 Query results' documentation may have matched well with the query, they do not match as well with the Golden Code. One way to bridge this semantic gap is by creating relationships between source code tokens and documentation tokens. See Section 4.1.

An open question is to understand the difference between the performance of this PLIR instantiation with respect to the different similarity ranking functions. Although the trade-offs between these similarity ranking functions are well-understood in traditional information retrieval systems, more research must be done to determine how they operate in this domain and whether the CCR@K is a meaningful measurement tool.

4. CANDIDATE AZRs

As mentioned in Section 2.4, as a result of the structure of the DocDB and the information stored therein, a modular set of analyses can be performed to improve baseline retrieval performance. This section describes a candidate AZR and evaluates its performance.

4.1 Data flow AZR

A data flow analysis can be performed on the methods and its associated source code to develop an extended WordNet-like relationship graph between words. By studying variable assignments in a method, it is possible to determine word relationships that are unique to particular projects. For instance, in

a project that implements a user interface, data flow analysis might reveal that “window” and “button” are related even though they are not synonymous in everyday English.

4.1.1 JavaFlow

JavaFlow is a proof-of-concept implementation of such a data flow AZRs for the DocDB for methods written in Java. JavaFlow iterates through the assignment statements of every method associated with a particular package. The name of the assignment's lvalue is deemed to relate (somehow) with each of the variables referenced by the code computing the rvalue. For example, JavaFlow considers that `b` and `c` are related to `a` in `a = b + c;`. JavaFlow leverages Java programming style and expands those relationships based on Camel Case tokenization. Therefore, JavaFlow considers `apple` and `fruit` to be related to `skin`, `color`, `fresh` and `taste` in `appleFruit = skinColor + freshTaste;`.

When used to analyze the code from the open source OkHttp library, for example, JavaFlow determines that `cache` is related to `dir`, `disk`, `lru`, `version`, `open` and `app` while `connection` is related to `url`, `client`, `open` and `server`.

4.1.2 JavaFlow Evaluation

To evaluate JavaFlow's affect on retrieval and ranking performance, the experiment described in Section 3 was repeated and the queries were extended to include calculated word relations. Words JavaFlow deemed as related were considered to be synonyms. Up to three of the top synonyms (according to a rank based on the count of co-occurences) for each of the query words were added to the original query and the results were reanalyzed.

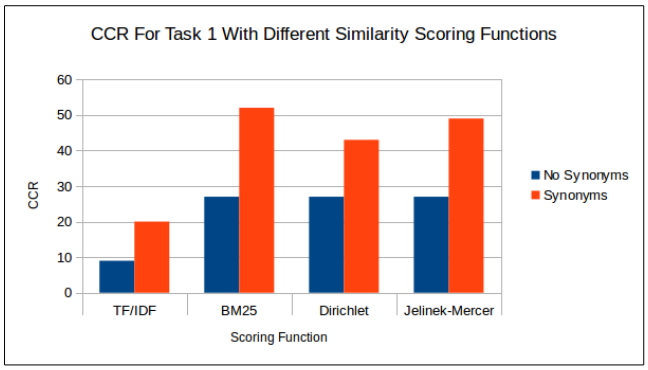

Figure 5: CCR for Task 1 with different similarity scoring functions

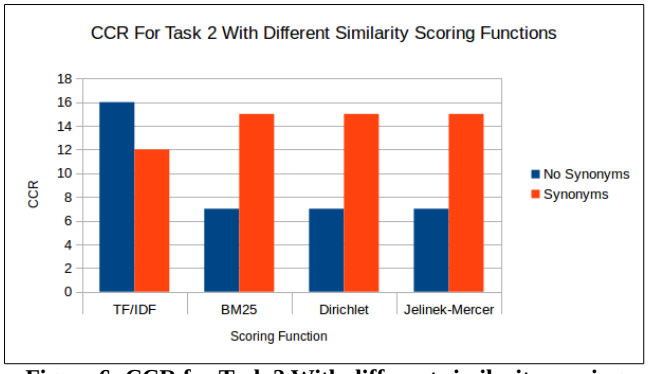

Figure 6: CCR for Task 2 With different similarity scoring functions

As Figures 5 and 6 show, JavaFlow improved performance of the PLIR system significantly. The increase of CCR@10 was accompanied by a subjective improvement in system performance: when using synonyms, the PLIR system was able to retrieve the exact code snippet for Task 1 (the recipes themselves are part of the source code repository for OkHttp).

4.1.3 Data flow AZR Improvements

JavaFlow is a very naïve implementation of a data flow AZR, yet the results are incredibly promising. Expanding the data flow AZR to other languages beyond Java is a logical next step. Additionally, data flow AZR is a perfect way to study the importance of more formal methods of programming language design and analysis on the effectiveness of a natural language source code information retrieval system.

5. FUTURE WORK

PLIR opens the door to considerable opportunities in research and development on natural language source code information retrieval systems.

5.1.1 Analyzers

In order to expand the types of source code available to researchers working with PLIR, additional Analyzers will be developed. Research on which languages are most popular will determine the order in which Analyzers are written. One analysis suggests a possible ordering: JavaScript, Ruby, PHP and Python. [6] As Analyzers are added, the additional source code that can be imported into the DocDB will give researchers more raw material to test their hypotheses.

More urgent than adding Analyzers is fully testing and debugging existing Analyzers. Notably, both the Java and C++ Analyzers must be extended to properly handle function polymorphism.

5.1.2 AZRs

Besides the future work described for data flow AZR (Section 4.1.3), there are many opportunities to extend PLIR through custom AZRs. Consider the opportunity for increasing/decreasing the rank of a particular result depending upon how often other code refers to it. References to a method may be invocations from other functions, but may also be derivative through the enclosing namespace. Because the DocDB contains the source code for methods and dependency relationships among classes, implementing such an AZR will be straightforward.

This is not the only type of AZR that could fit in PLIR. Another example is topic generation using latent Dirichlet allocation. [8] Topics generated for the source code and documentation in the DocDB using Linstead's method could be used to improve system performance.

Ultimately, however, because of the system's modularity, only a researcher's intuitions limit the possibilities.

5.1.3 Indexers, retrievers and rankers

Solr as the basis for PLIR indexing, retrieving and ranking provides many paths for future work. Solr's indexing methods can be customized based upon the document type. For PLIR as instantiated in Section 3, this means one type of index for code and another type for documentation. The indexing and tokenization scheme presented in Section 3.3 may not be appropriate for source code not written in Java. Moreover, not considered is how to index function parameters (their names and types).

Solr also provides the opportunity to experiment with different similarity ranking functions. Sections 3.4 and 4.1.2 gave examples of system performance using different ranking functions provided by Solr. In addition to researching which of these performs optimally, it may be worthwhile developing similarity ranking functions that are particularly suited to source code and program documentation.

Finally, other candidates should be considered for indexing, ranking and retrieving. One example is ElasticSearch. [3]

5.1.4 Evaluation metrics

In the future, it will become increasingly important for researchers to develop a standardized mechanism for evaluating the performance of natural language source code information retrieval systems. This type of metric will make it possible to quantitatively determine the effect on system performance with respect to researchers' hypothesis. This paper presents one option: CR and CCR@K. This method relies on a manual annotation of source code to determine the Golden Code and does not reflect the relative of position of documents within the first K results. Additional work must be done to incorporate the intuitions underlying the Discounted Cumulative Gain, F-Measure, etc evaluation metrics from the field of information retrieval.

6. CONCLUSION

This paper presented PLIR, an extensible system for developing, researching and deploying natural language source code information retrieval systems. It described PLIR's basis in techniques and systems from the fields of programming language design and analysis and information retrieval. It showed how PLIR combines existing tools in a way that makes it easy for developers and researchers to study and enhance each of the subcomponents that comprise a functional natural language source code information retrieval system. Additionally, this paper described an implementation of PLIR and showed how it can be used to evaluate hypothesis about improvement on existing information retrieval techniques related to natural language processing of computer source code.

7. REFERENCES

[1] About the Black Duck Open Hub, n.d. Retrieved on 26 Sept. 2014, from Black Duck Open Hub Blog: http://blog.openhub.net/about/.

[2] Russ Cox. Regular Expression Matching With a Trigram Index or How Google Code Search Worked, Jan. 2012. Retrieved on 26 Sept. 2014 from Russ Cox blog: http://swtch.com/~rsc/regexp/regexp4.html.

[3] Elasticsearch.org Open Distributed Real Time Search & Analytics, 2014. Retrieved on 3 Dec. 2014, from Elasticsearch home page: http://www.elasticsearch.org/.

[4] Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE '09). IEEE Computer Society, Washington, DC, USA, 232-242. DOI=10.1109/ICSE.2009.5070524 http://dx.doi.org/10.1109/ICSE.2009.5070524

[5] Krugle OpenSearch, n.d. Retrieved on 26 Sept. 2014, from krugle – software development productivity: http://krugle.com/krugle-opensearch.php.

[6] Language Popularity on GitHub, 3 May 2014. Retrieved on 3 Dec. 2014, from Otaku, Cedric's blog: http://beust.com/weblog/2014/05/03/language-popularity-on-github/.

[7] Latest News, 10 Nov. 2005. Retrieved on 29 Sept. 2014 from Codase News: http://www.codase.com/news.html.

[8] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. 2007. Mining concepts from code with probabilistic topic models. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (ASE '07). ACM, New York, NY, USA, 461-464. DOI=10.1145/1321631.1321709 http://doi.acm.org/10.1145/1321631.1321709

[9] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. 2004. An Information Retrieval Approach to Concept Location in Source Code. In *Proceedings of the 11th Working Conference on Reverse Engineering* (WCRE '04). IEEE Computer Society, Washington, DC, USA, 214-223.

[10] Merbobase Source Code Data Sets, Dec. 2012. Retrieved on 26 Sept. 2014, from the Software Engineering group at the University of Mannheim, Germany: http://merobase.informatik.uni-mannheim.de/sources/.

[11] Masaru Ohba and Katsuhiko Gondow. 2005. Toward mining "concept keywords" from identifiers in large software projects. *SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1-5. DOI=10.1145/1082983.1083151 http://doi.acm.org/10.1145/1082983.1083151

[12] OpenGrok, n.d. Retrieved on 29 Sept. 2014, from OpenGrok website: http://opengrok.github.io/OpenGrok/.

[13] OkHttp, 2014. Retrieved on 3 Dec. 2014, from OkHttp website: http://square.github.io/okhttp/.

[14] Rosetta Code, 5 Aug. 2011. Retrieved on 26 Sept. 2014, from Rosetta Code: http://rosettacode.org/wiki/Rosetta_Code.

[15] Scott Grant and James R. Cordy. 2010. Estimating the Optimal Number of Latent Concepts in Source Code Analysis. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation* (SCAM '10). IEEE Computer Society, Washington, DC, USA, 65-74. DOI=10.1109/SCAM.2010.22 http://dx.doi.org/10.1109/SCAM.2010.22

[16] searchcode | source code search engine, n.d. Retrieved on 26 Sept. 2014, from searchcode: http://www.searchcode.com.

[17] Giriprasad Sridhara, Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2008. Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension* (ICPC '08). IEEE Computer Society, Washington, DC, USA, 123-132. DOI=10.1109/ICPC.2008.18 http://dx.doi.org/10.1109/ICPC.2008.18

[18] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering* (ICSE '11). ACM, New York, NY, USA, 101-110. DOI=10.1145/1985793.1985808 http://doi.acm.org/10.1145/1985793.1985808

[19] Solr Features, 2014. Retrieved on 3 Dec. 2014, from Solr website: http://lucene.apache.org/solr/features.html.

[20] Story of OpenGrok, 4 Mar. 2013. Retrieved on 26 Sept. 2014, from OpenGrok/OpenGrok Wiki: https://github.com/OpenGrok/OpenGrok/wiki/Story-of-OpenGrok.

[21] UCI Source Code Data Sets, 2010. Retrieved on 26 Sept. 2014, from website of Cristinia Videira Lopes: http://www.ics.uci.edu/~lopes/datasets/.