A Case Study of Automated Feature Location Techniques for Industrial Cost Estimation

Ameer Armaly*, John Klaczynski†, and Collin McMillan*
*Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN 46615
Email: {aarmaly, cmc}@nd.edu
†SimVentions, Inc.
Fredericksburg, VA 22408
Email: jklaczynski@simventions.com

***Abstract*—We present a case study of feature location in industry. We study two off-the-shelf feature location algorithms for use as input to a software cost estimator. The feature location algorithms that we studied map program requirements to one or more function points. The cost estimator product, which is the industrial context in which we study feature location, transforms the list of function points into an estimate of the resources necessary to implement that requirement. We chose the feature location algorithms because they are simple to explain, deploy and maintain as a project evolves and personnel rotate on and off. We tested both feature location algorithms against a large software system with a development lifespan of over 20 years. We compared both algorithms by surveying our industrial partner about the accuracy of the list of function points produced by each algorithm. To provide further evidence, we compared both algorithms against an open source benchmarking dataset. Finally, we discuss the requirements of the industrial environment and the ways in which it differs from the academic environment. Our industrial partner elected to use Lucene combined with the PageRank algorithm as their feature location algorithm because it balanced accuracy with simplicity.**

I. Introduction

Feature location is the task of mapping software requirements to the code that implements those requirements [1], [2]. Feature location is very important to the software development process. A developer newly assigned to a project might use feature location to find the code relating to a defect or change request. Even an experienced developer might use feature location to find the starting point for a defect or change request because the relevant portion of the project has changed so drastically that his previous understanding of it is no longer applicable. Dedicated feature location algorithms that take into account the structure and semantics of the project are more useful than simple pattern matching.

Academic research has produced a wide variety of feature location approaches (see section III). Static approaches examine the structure of the source code to find code relating to a feature. Dynamic approaches examine the program as it runs to trace the exact functions called by a feature. Textual approaches employ textual analysis techniques to find the code relating to a feature. Finally, hybrid approaches combine two or more of the above approaches to produce a new approach.

In this paper, we study the application of feature location in industry. The context of our study is a product for software cost estimation. Software cost estimation is the task of approximating the time or monetary resources that a software product will consume during development [3], [4]. In this context, a cost estimator product takes the list of locations in code returned by a feature location algorithm and translates it into an estimation of the resources required to implement that feature. The larger the number of affected locations, the more complex the requirement. The cost estimator measures the affected locations in terms of function points. The role of the feature location component is to match requirement text to a list of relevant function points to be passed to the cost estimator. A function point is a location in a program that represents some quantifiable amount of effort. Typically, a function point is one of an "input", "output", "master file", "interface", or "inquiry." For example, an input function point in a program might be an integer that is passed as an argument to the program. That integer would be counted as a fixed number of work units. A more complex input, such as a string, would be counted as a higher number of work units. Function points are a superior measure of software size than source lines of code (SLOC) [5], [6], [7], [8]. We discuss function points and cost estimation in greater detail in the Background Section IV-A.

Our industrial partner is tasked with developing and maintaining programs of varying size and complexity for the United States Department of Defense. Choosing a feature location algorithm for this environment requires balancing accuracy with simplicity and ease-of-deployment. As programmers rotate on and off the project they will need to be able to learn and use the algorithm and explain its results to their superiors. The algorithm should be able to deal with changing requirements as it will be used to estimate the resources required for proposed changes. The algorithm should be usable in the field to estimate the effort required to alter a program currently in use when unforeseen challenges or use cases arise. This contrasts with the academic research environment where the goal is to advance the state of the art by devising an algorithm with better precision and recall. In Section VIII, we expand

on these differences between academic and industrial use of feature location, and discuss our lessons learned.

We report on our experience adapting two feature location algorithms to a large legacy United States Army system written in Ada and having a development lifespan of over 20 years. This system runs on specialized hardware, making dynamic feature location algorithms impractical as they require collecting execution traces every time the algorithm is to be run. Therefore we chose two textual/static feature location approaches. The first algorithm used Lucene to map requirement text to function points. The second algorithm used Lucene in combination with the PageRank algorithm [9]. We present our algorithms, our experiences using the algorithms including the requirements and concerns of our industrial partner, and our recommendations for others with similar needs. Put briefly, we found that combining Lucene with PageRank provided the best balance between good performance and simplicity of deployment.

II. The Problem

We study the problem of applying feature location in an industrial setting, in the context of a cost estimation product. The cost estimation product is able to predict the expense of different components in code. However, it has a drawback in that it is not able to predict the expense of different *features*. This drawback is serious for some organizations, notably the U.S. Navy [10], because features can change rapidly over the lifetime of a software system. Cost estimations on one version of the software will not necessarily apply to the next.

The Navy currently relies on measures such as SLOC to estimate the time and effort required for a project. SLOC varies between languages, even languages in the same family such as Java and C++. Estimating the SLOC that will be introduced into a project by a new feature ultimately requires a subject matter expert (SME). A SME is not only someone who is familiar with the problem domain, but also someone who is familiar with the code. A SME must manually analyze the current state of the project and the new requirements and estimate the number of SLOC that will be introduced. The resulting estimates are varied and ultimately subjective. Different SMEs can disagree on the amount of effort required for a change. SMEs are likely to know certain components of the project better than others. A SME might have an outdated understanding of a particular component because it has recently changed. This creates a risk for project managers who are tasked with avoiding cost overruns while making sure a program can fully implement its requirements.

Function points provide an alternative metric that is more stable than SLOC. Calculating function points automatically eliminates the need for SMEs to estimate the cost of features. The number of function points changes more slowly as the program evolves. This provides a more accurate picture of the changes in the program's complexity since it measures the complexity of the changed code in terms of its interaction with the rest of the program rather than strictly in terms of its size. Function points are language-neutral, allowing accurate complexity analysis of projects with components written in multiple languages. Finally, function points provide a more high-level description of the complexity of a program or a feature. They quantify the interactions between the project components and provide a stable metric of project size. Function points do not, however, provide a ready means of grouping existing function points by feature, nor do they provide a means of estimating how many new function points will be introduced by a feature without the judgment of a SME.

Feature location has the potential to solve this problem. Feature location techniques produce a mapping of features to source code. Features can be described verbally or by the actions taken by the user. In the first case, the text of a feature description is analyzed to find areas in the code that match keywords in the text. A simple case of this approach is using the grep Unix utility to search for occurrances of the keywords in the code. In the second case, the program is run while being monitored and the code that runs in response to a user action such as a mouse click is mapped to that feature. The mapping produced by a feature location algorithm can be used to assign costs to features: the cost of a feature is based on the cost of the code to which the feature is mapped.

But there are many automated feature location techniques available, and the performance of these techniques varies in different domains. Some techniques are language-specific. Other techniques require the user to run the program and execute the action that they want to map to code. Other techniques are costly to implement either because they are intrinsically complex or because they combine several approaches. In this paper, we explore two of these techniques for the purpose of cost estimation in Navy systems. Specifically, we seek to map each requirement in a requirements document to a set of function points. The function points can then be used by the cost estimation product in order to estimate the time and effort required for that feature. We compare the results of each algorithm by having a SME rate the relevance of the top ten function points returned by each algorithm. The results of our study will improve the accuracy of the cost estimation by helping to ensure that the feature location mapping is correct. These results could also benefit designers of cost estimation products in other domains.

III. Related Work

Feature location techniques can be classified as static, dynamic, textual, or hybrid. Biggerstaff *et al.* [1] formulated the concept assignment problem and the first static feature location approach. Chen and Rajlich [11] devised an approach based on examining the Abstract System Dependence Graph to find material relevant to a feature. Robillard [12] improves upon this approach by grouping functions by specificity and reenforcement. Marcus *et al.* [13] compared the available static feature location techniques.

Dynamic approaches examine the execution of a program to determine what parts are activated by a specific feature. Software reconnaissance [14] compares two execution traces

of a program: one with the target feature active and one with the target feature inactive. Eisenberg and De Volder [15] improve upon this approach by scoring code elements on their relevance to a feature rather than using binary judgements. Wong emphet al. [16] examine execution slices of test cases to determine the full scope of a feature in source code rather than finding a likely place to start.

Marcus *et al.* [17] introduced textual analysis by applying Latant Semantic Indexing (LSI) [18] to source code. This technique was extended [19] to allow the user to mark certain results as relevant or irrelevant to more accurately frame the query. Grant *et al.* [20] applied Independent Component Analysis (ICA) to feature location. Lukins *et al.* [21] use use latent Dirichlet allocation (LDA) [22] to find methods that relate to a bug.

Hybrid approaches combine the results of two or more approaches to produce a more accurate final result than any of their individual components. Promesir [23] combines LSI with a scenario-based probabilistic ranking by using an affine transformation. SITIR [24] uses LSI to filter the functions returned by examining an execution trace. Revelle *et al.* [25] combine SITIR with web mining to add dependency information and achieve a better result. Dit *et al.* [26] used web mining and execution traces to filter the results produced by LSI. Their approach could use either the HITS [27] or PageRank [9] algorithms.

Many feature location approaches must be manually tuned to achieve the best result. Biggers *et al.* [28] examine five configuration parameters and their impact on LDA systems. Panichella *et al.* [29] automatically tune LDA for traceability link recovery, feature location and software artifact labeling using a genetic algorithm.

IV. Background

In this section, we provide background on the function point analysis that serves as context for our study. We also discuss the two feature location algorithms that we test later. Note that we discuss function points in detail because we use the feature location techniques to match requirements text to function points in source code (see Section IV-B for details).

A. Function Point Analysis

A *function point* is a location in software that represents a measurable cost to create. Function points are calculated at the borders of software components, e.g., among the classes in a Java package or among the programs in a software system. Different types of function points represent different costs. As detailed in a well-cited study by Matson *et al.* [7], the typical types of function points are:

- **Inputs**, which denote data that travel into a software component, e.g., a parameter of a method inside a class that is called by a method in a different class.
- **Outputs**, which denote processing or creating data that then travel out of a component, e.g., a value that is returned from a method in one class to a method in a different class.
- **Inquiries**, which are identical to outputs except that no alterations to the data occur, e.g., returning a value that is read from a file but to which no modifications are made.
- **Internal Master Files**, which are sets of logically-related data in a component.
- **External Interface Files**, which are internal master files of other components upon which a component depends.

Each of the types of function points are categorized as low, medium, or high complexity. Then, weights are applied to the types and levels. For example, a low-complexity input might be counted as one "work unit", while a high complexity output might equal five work units. The total count of work units is governed by the formula.

$$F = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} * z_{ij} \quad (1)$$

F is the work unit count, z_{ij} is the number of function points of a particular type at one complexity level, and w_{ij} is the weight assigned to the function points of that type and complexity. For example, z_{13} would represent the number of input function points at high complexity, and w_{22} would be the weight assigned to output function points at medium complexity.

Function points can be calculated manually or automatically. The manual process involves a human expert who reads requirements and design documentation in conjunction with source code. The expert then marks each function point in the documents, and computes the total function point count at the end. Some experimental tools can calculate function points based on design documents [30], though the state-of-the-practice is that automated solutions focus on analyzing source code. For example, the Object Management Group procedures [31] emphasize analysis of code. The five function point types and three complexity levels tend not to be different for the automatic and manual approaches, though variations have been proposed [32], [33].

Function points were designed in response to the use of Source Lines of Code (SLOC) as measure of software size [34]. The perception is that SLOC is not a dependable metric because 1) it is highly dependent on language, and 2) different lines can have very different costs even in the same program [35]. A transition from SLOC to function points has been proposed for decades [36], but a lack of standardization and a large amount of legacy software have prevented some organizations from adopting function points until recently. The context of this paper is a transition from SLOC to function points at the U.S. Navy [10].

B. Feature Location Techniques

Feature location is the task of matching descriptions of the behavior of software to the source code implementation of that behavior [2]. Feature location is an active research area, and hundreds of variations, improvements, and domain-specific enhancements have been proposed. These different techniques can be placed on a spectrum from text-based to

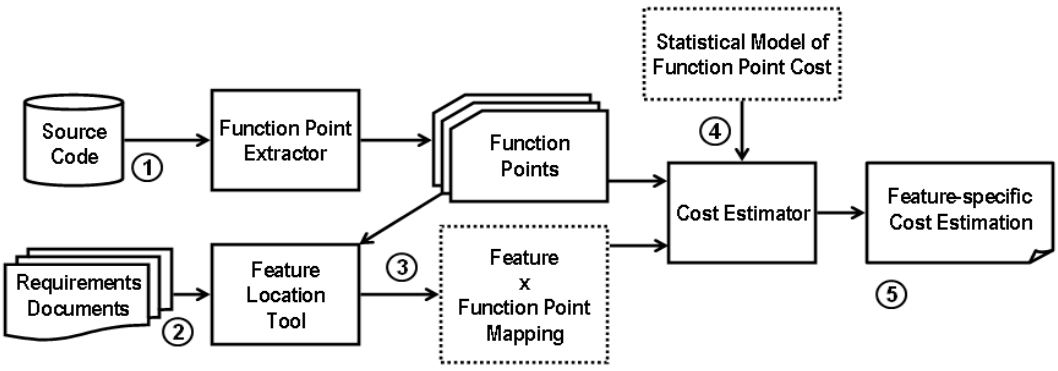

Fig. 1. Overview of the context in which we evaluate the different feature location techniques. The feature location tool (area 2) is part of a product that estimates the cost of features. The cost estimation itself is out of the scope of this paper; our goal is to choose the feature location tool for the product. Solid boxes indicate components that we created for our approach. Dashed lines indicate intermediate data. Section V describes this figure in detail.

structural-based. Text-based refers to techniques that match the keywords from natural language documentation to identifier names and comments in source code. In contrast, structural-based techniques use clues from the structure of the source code such as function calls and inheritance. A majority of techniques combine both textual and structural information [2].

The remainder of this section describes two techniques for feature location that we study in this paper. These techniques have been proposed and evaluated elsewhere, therefore we provide only a high-level explanation of each. We chose these techniques because they are widely-used in industry or widely-cited in literature.

1) Lucene Comparison: Lucene is a Java implementation of the Vector Space Model (VSM) used for calculating text similarity - a pure text-based approach. VSM is a representation of the vocabulary of a set of documents. Every document is represented as a vector. The contents of the vector are the words that appear in the document. Each position in the vector is a positive integer that indicates the number of times the word appears. To be adapted as a feature location tool, the "documents" are typically classes or methods in a software program. The "words" are identifier names and keywords from the comments in those classes or methods. Thus, the VSM generated for source code is a set of vectors, where each vector represents one class or method, and each position in the vector is the count of occurrences of one identifier or word from a comment. However, the counts of occurrences in each vector are modified using the Term Frequency/Inverse Document Frequency (*tf/idf*) formula.

$$tf = \frac{n}{\sum_k n_k} \tag{2}$$

In this formula, n is the count of occurrences of the word in a document, and $\sum_k n_k$ is the total number of occurrences of the word in all documents. The idea behind the tf/idf formula is that words that occur frequently in many documents are weighted less than words that occur in only a few documents.

When Lucene is used as a feature location technique, requirements documents are compared to components of source code. For example, a description of one requirement in a program is matched to the classes or methods in the code. The requirement is modeled as a vector of words, and the positions in the vector are weighted using tf/idf. Then, the similarity between the requirement and each class/method is calculated using the cosine similarity between the requirement's vector representation and the vector representation of each class or method. The classes/methods with the higher similarity values are considered more likely to implement that requirement.

2) Lucene with PageRank: The second feature location technique is a variation of the first. We apply the PageRank algorithm [9] to produce weights for the function points in a project. PageRank calculates the relative importance of an entry in a set of documents which in this case are the functions in the call graph of the project. For some function p, the initial value is $\frac{1}{N}$, where n is the number of functions in the project. The algorithm then sums the PageRank values for all functions that call p and adds them to the initial value. The algorithm can be expressed as follows:

$$PR(p) = \frac{1}{N} + \sum_{j \in I(p)} \frac{PR(j)}{|O(j)|} \tag{3}$$

N is the number of functions in the project, textit$i(p)$ is the number of functions that call the function p and $o(p)$ is the number of functions called by p. The algorithm must be run until the values converge. We use PageRank to assign weights to each function in the call graph of the project. The weight of each function is then multiplied by the Lucene score for any function point occuring in that function to push more important functions higher in the list of search results. PageRank has been used as a component of previous feature location approaches. Portfolio [37] is a code search engine that outperforms Google Code Search at both finding relevant functions and finding use cases for those functions. Dit *et al.* [26] combined PageRank with dynamic analysis and information retrieval to produce a feature location model that outperformed its component techniques.

V. The Context of the Case Study

The context of our study is a software cost estimation product. Figure 1 portrays this context. While the specifics of the cost estimation technique are proprietary and beyond the scope of this paper, we provide an overview in this section to explain our usage scenario of the feature location tools. Our study results and recommendations are based on this scenario.

The inputs to the product are 1) the source code, and 2) the requirements documents associated with a software program. The source code is written in Ada. The requirements must be written in English, and in a format that delineates different feature descriptions (e.g., one feature description per line, or per section header). The output of the product is a cost associated with each feature. Changes to the requirements documents, such as the addition of a feature, result in a cost based on the feature costs.

The internals of the product work generally as follows. A *function point extractor* process the source code (Figure 1, area 1) to determine the function points in the program. This extractor is based on the Object Management Group standard for automated function point analysis [31]. The function points it locates fall into the categories laid out in Section IV-A. For the version of the product in our study:

- **Inputs** are parameters to function calls into a class.
- **Outputs** are return values from function calls into a class.
- **Inquiries** are not distinguished from outputs (e.g., return values were not distinguished as modified).
- **Internal Master Files** are all source files that contained functions.
- **External Interface Files** are not included because there were none defined by the application.

All weights are equal to one. That is, the version of the product in our study does not distinguish among low, medium, or high complexity function points.

The *feature location tool* reads the requirements documents (Figure 1, area 2) and matches them to function points (Figure 1, area 3). Note that the feature location tool actually maps feature descriptions to functions (see Section VI-B). We translate the functions to the function points by giving the similarity value for the function to the function points in that function. For example, consider a result from the feature location tool where feature A is similar to function F with a value of 0.89. If function F has two parameters that are function points, then the product returns those function points as having a similarity value of 0.89 to feature A. While it is typical in academic research to connect requirements to functions or classes, our industrial partner needed the connections to function points for purposes of cost estimation. This need meant that we had to translate connections from functions to function points.

No text preprocessing is performed by the product for the function points or the documentation. But, each feature location technique may perform its own preprocessing, including stemming, splitting, and stop word removal. We use the default configuration of the tools as described in the related work, or as available when downloaded.

The *cost estimator* uses a statistical model to predict the cost of each function point (Figure 1, area 4). This procedure is similar to the state-of-the-art in cost estimation (see Section III), where the statistical model is created from records of previous experience. Function points (and combinations of function points) are known to have taken a concrete amount of time, or to have required a specific number of programmers, on earlier projects. The cost estimator links statistical averages about these function points to the function points for each feature. Then, the cost of each feature is estimated based on these averages (Figure 1, area 5).

VI. Case Study Design

This section describes the design of our case study, including our research questions, our methodology for answering those questions, and other details about the study environment.

A. Research Questions

Our research objective is to determine which feature location technique should be used in our industrial context of cost estimation. Performance is a key factor, but ease-of-use is also an important practical concern for deployment. Any increase in complexity of the solution must be justified by a worthwhile increase in performance. As such, we pose the following Research Questions (RQs):

- RQ_1 Which of the two feature location techniques has the highest performance, as perceived by the industrial partner?
- RQ_2 Which of the two feature location techniques has the highest performance when evaluated against an open source benchmarking data set?
- RQ_3 Which of the two feature location techniques has the lowest cost to implement, as perceived by the industrial partner?
- RQ_4 Which of the two feature location techniques achieves the best balance between performance and cost?

The rationale behind RQ_1 is that quality of the results will vary for each of the feature location techniques, and that the industrial partner will observe these variations. The degree of these variations will be a key factor in the decision to deploy the techniques. Therefore, we collect information about the quality of the results from the techniques. To supplement these observations we test both techniques against an independent mapping of requirements to functions in RQ_2. Likewise, cost-to-implement is a factor in the decision to use a technique, so we determine the industrial partner's perceptions of this cost through RQ_3. Note that cost-to-implement may depend on internals of the feature location technique, such as whether the technique requires structural information from the source code. This structural information may, for example, be cost-prohibitive to obtain. Cost-to-implement must be balanced with performance to form a decision about which technique to use. We study this balance in RQ_4.

B. Methodology

The methodology of our study is to 1) implement the two techniques in the context of the cost estimation tool, 2) test both techniques against an benchmarking dataset and 3) to survey the industrial partner based on the experience using these different techniques. Note that since this study is a component of an ongoing commercial project, our survey must not be intrusive to the industrial partner, and it must not slow development. Thus, a standard cross-validation user study involving dozens of programmers is not viable. Instead, we design the survey such that the industrial partner is blind to the techniques he evaluates, to avoid a potential bias.

The survey design for RQ_1 is as follows. First, we execute each feature location technique to obtain a top-ten list of *function points* in the source code that map to each *feature* described as text in the requirements documents. Next, we built an online survey that presents a feature description to the industrial partner, along with function points from the source code. The function points are the same function points in the top-ten lists from the feature location tools, but the survey does not indicate which function point was recommended by which technique. The survey then allows the industrial partner to select whether or not the function is relevant to the feature.

The survey continues showing feature descriptions and function points to the partner until all features are evaluated. However, due to time constraints we ask the partner to exit the survey after 30 minutes.

We answered RQ_2 by testing both algorithms against a mapping of requirements to functions for an open source benchmarking dataset. The dataset does not map requirements to function points so we are forced to map requirements to functions rather than function points in this evaluation. The requirements were taken from the JavaScript specification. We ran both algorithms on all requirements in the specification. For each requirement we extracted a set of relevant functions. Each function had a score that was returned by Lucene. We normalized this score by dividing the score by the score of the highest-scoring function. We filtered all functions with a normalized score of less than 0.5. The minimum score is adjustable. We use SRCML [38] to creat an XML representation of the source tree. We then parsed the SRCML representation to extract individual functions and to generate the call graph for use by PageRank.

We used a custom Lucene analyzer for the requirement text and another analyzer for the java source code. Both analyzers filtered whitespace and punctuation tokens. Both analyzers also filtered out english stop words such "a" and "the". The source code analyzer filtered out Java stop words such as "int" and "if." The requirement text analyzer did not. We measured two metrics: precision and recall. Precision is the percentage of correct functions returned by the algorithm. Recall is the percentage of all correct functions for the requirement that the algorithm returned. We filtered out empty requirements from both the benchmarking dataset and the returned dataset. Empty requirements include sections that have no text themselves but have several subsections.

We use a brief qualitative interview approach [39] to answer RQ_3. We ask the industrial partner questions related to the inputs of the feature location techniques (e.g., textual or structural data), to determine the perceived difficulty of obtaining those inputs for the typical use case of the cost estimation tool. In addition, we ask questions related to the difficulty of the specific tools. For reproducibility, we provide the exact list of questions in Table I. However, note that we used these questions as a starting point for conversation, and do not limit our qualitative results to specific answers to these questions.

The chief evidence to answer RQ_4 is the decision that the industrial partner makes regarding the feature location technique. We provide discussion and recommendations related to this decision based on the qualitative interview process.

C. Subject Application

The subject application of the survey is a control program for an aviation training simulator. The program is written in Ada. It includes over 200K SLOC in 253 files and over 3000 functions. The software has a development and usage history spanning over 20 years. The system is not classified, but is closed-source and confidential.

The subject application for the benchmarking dataset is the open-source Rhino JavaScript engine [1], developed by the Mozilla Foundation. Rhino is is typically used to add JavaScript functionality to applications and is the default JavaScript engine for J2SE 6. It is written in Java and contains over 6K SLOC in over 1300 functions and 129 files.

D. Threats to Validity

Like any study, this work contains possible threats to validity. One threat is the subject application. Different results are possible on different systems. We attempted to mitigate this threat by using a representative example of the programs for which the cost estimation tool will be used. Unfortunately, due to legal and practical constraints, we were unable to conduct the study on additional subjects. However, as research by our industrial partner continues, we believe we can verify our finding against additional subjects.

Another threat is that we only used one benchmarking dataset to support our conclusions. The dataset mapped requirements to functions rather than function points. Other

[1] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

TABLE I
SURVEY QUESTIONS

	Question
1	What is your opinion on the difficulty extracting, and availability of, text data in the source code?
2	Would you expect to be able to extract structural information for all source code you use as input to the cost estimator?
3	Would you expect to be able to compile all that source code?
4	Did you encounter any unexpected challenges in building the text corpus for the techniques?
5	Given what you know now, which feature location technique do you expect to use for the cost estimator?

datasets or a dataset that mapped requirements to function points could yield different results. We used a statically-generated call graph when applying PageRank to this dataset. This call graph is potentially incomplete. It may not contain indirect function calls that come from outside the program itself. The number of calls from one method to another may be governed by the nature of the input supplied to the program. We discuss indirect function calls in more detail in section VIII-C.

Finally, only one expert from the industrial partner was able to participate in the study. Other experts may have different conclusions. Nevertheless, our study depicts an actual scenario, and the reality is that one expert may be relied upon to make key decisions. Our caveat is that the results of our study may vary depending on setting; the intent of this study is to provide an example of one experience using feature location in practice.

VII. Evaluation Results

In this section, we present the results of the evaluation of our approach.

A. RQ_1: Relevance of Function Points

We found statistically-significant evidence that the combination of Lucene and PageRank produced more relevant function points than Lucene alone. The average relevance score for the combination of Lucene and PageRank was 2.7 where as the average score for Lucene alone was 2.37. To determine statistical significance, we used the Mann-Whitney test by posing hypothesis H_1:

H_1 The difference between the relevance of the results produced by combining Lucene with PageRank and using Lucene alone is not statistically significant.

We rejected this hypothesis based on the results of the Mann-Whitney test shown in Table III. An α value of 0.05 means that the difference between the samples was statistically significant; the results produced by combining Lucene and PageRank were more relevant by a statistically significant margin.

B. RQ_2: Benchmarking Dataset

We found that combining Lucene with PageRank increased precision while decreasing recall when compared to Lucene alone on the independent data set. Lucene alone returned a precision of 53.5% and a recall of 0.8%. Lucene with PageRank returned a precision of 75.5% and a recall of 0.42%. To shed more light on why recall is low we reran the experiment with multiple minimum confidence scores between 0.5 and 0 to determine how the relevant functions were distributed. The following table hosws the results.

Recall increases as the minimum score decreases, but the largest increase occurs between minimum scores of 0.05 and 0. This suggests that many of the relevant functions score very poorly with Lucene. Applying PageRank promotes certain relevant functions but leaves many behind.

C. RQ_3: Cost of Implementation

We found that both techniques were easily deployable by the industrial partner. The industrial partner believed that the most difficult aspect was placing the extracted text data in context, e.g., determining if a package is a utility package as opposed to project-specific code. The industrial partner believed both techniques could be generalized to a wide range of projects.

D. RQ_4: Cost / Performance Balance

We found that the combination of Lucene and PageRank gave the most relevant results while imposing minimal overhead. The PageRank algorithm can be run once and the data reloaded as many times as necessary. The industrial partner chose to use this combination for cost estimation.

VIII. Conclusions & Lessons Learned

In this section we discuss the distinct requirements as well as potential pitfalls of using an approach of this kind in an industrial as opposed to an academic environment.

A. Industrial Requirements

We identified four key industrial requirements for feature location approaches separate from the usual requirements present in an academic research setting. We derive these requirements from our observations and discussions with our industrial partner. This list is not exhaustive but will hopefully provide guidance for others seeking to use feature location in industry.

It was not practical for us to use dynamic feature location approaches for this project. These approaches depend on

TABLE II
Results for a range of minimum score values.

Method	Minimum Score	Precision	Recall
Lucene	0.50	53.56%	0.88%
Lucene+PageRank	0.50	75.24%	0.43%
Lucene	0.45	52.21%	1.12%
Lucene+PageRank	0.45	71.25%	0.52%
Lucene	0.40	49.89%	1.46%
Lucene+PageRank	0.40	67.92%	0.62%
Lucene	0.35	47.56%	1.91%
Lucene+PageRank	0.35	64.01%	0.79%
Lucene	0.30	46.48%	2.56%
Lucene+PageRank	0.30	59.31%	1.04%
Lucene	0.25	44.65%	3.41%
Lucene+PageRank	0.25	54.44%	1.40%
Lucene	0.20	41.84%	4.62%
Lucene+PageRank	0.20	50.71%	2.01%
Lucene	0.15	39.63%	6.50%
Lucene+PageRank	0.15	46.68%	3.03%
Lucene	0.10	37.32%	9.65%
Lucene+PageRank	0.10	42.88%	4.96%
Lucene	0.05	33.60%	15.88%
Lucene+PageRank	0.05	38.10%	9.54%
Lucene	0.00	28.44%	43.42%
Lucene+PageRank	0.00	28.38%	43.32%

TABLE III
STATISTICAL SUMMARY. MANN-WHITNEY TEST VALUES ARE U, U_{expt}, AND U_{vari}. DECISION CRITERIA ARE Z, Z_{crit}, AND p.

RQ	Metric	H	Approach	Samples	$\tilde{x}$	μ	Vari.	U	U_{expt}	U_{vari}	Z	Z_{crit}	p
RQ_1	Relevance	H_1	Lucene with Page Rank	99	3.000	2.707	1.373	5163	4455	129005	1.971	1.645	0.024
			Lucene	90	2.000	2.378	1.788						

running the program and comparing execution traces to determine which code executes when a given feature is enabled. These approaches are complex because they rely on examining a running program which often requires support from the underlying hardware. In this case the underlying hardware was a twenty-year-old aircraft simulator for which we do not have documentation. Moreover, the need to regularly rerun the feature location algorithm as the project changes and new requirements are proposed makes it impractical to have a dedicated engineer constantly running program traces on the aircraft simulator Therefore we were restricted to textual and static feature location approaches..

It should be possible to explain the algorithm to new programmers who are responsible for using it and reporting on its results. Since programmers rotate in and out of industrial positions this algorithm will likely be in place long after the people who chose to use it have left. It should be possible to explain the algorithm to project managers who will use the data returned by the algorithm to prioritize work and make other decisions relating to the project. Finally, auditors and investigators who are omnipresent in the public sector will need to be able to understand the algorithms and the decisions resulting from its output in order to assess whether those decisions were correct. This does not mean that industrial environments are constrained to only the most basic algorithms but it does mean that potential users should be aware that they will need to explain the approach on a regular basis.

The use case for our approach involves estimating the complexity of new requirements and changes to proposed or existing requirements rather than simply dividing the program up into logical components. This means that the algorithm must run in a timely manner as it will be run often. It is acceptable to obtain a "worse" result if it means that the approach runs faster. This requirement also means that the algorithm must not need to be manually tuned as the project requirements change.

As time goes on and a project evolves certain components may be written in different languages. Occasionally the whole project will be rewritten to take advantage of a new language or environment. The algorithm our industrial partner chose needs only a call graph to be able to generate scores. It does not need to run the program and examine it while running. It does not need to parse the source code beyond generating a call graph and extracting the bodies of functions. It also gains an advantage in portability from being written in JAVA; it can be applied in any environment with a JVM. This is not to say that our approach can be run on any environment as-is, only that it can be done with a small of effort. Much of the work can be done by reusing other tools or code.

B. Precision and Recall

The algorithms we tested returned high precision but very low recall when run against the open source benchmarking dataset. This means that a large percentage of the results were relevant to the requirement, but that there were a large number of relevant results that our approach did not return. This effect could be partially attributed to issues with the call graph such as indirection and outside function calls not being recognized. Another possibility is that the functions in the program are not similar to the requirement text because the functions concern the internal operations of the program whereas the requirements document we use is a specification separate from the program and produced by a different party.

Existing research places more importance on recall than precision [40], [41]. In spite of these issues our industrial partner chose to use the algorithm with the higher precision and lower recall. This algorithm is still being used for feature location today. The reason is that the cost estimates that will ultimately result from these algorithms will be used to create budgets and to allocate resources for future projects. Resources can include specialists such as programmers or testers as well as specialized equipment for manipulating or testing certain features of a project. These resources tend to be more expensive to allocate than programmers in the open market or off-the-shelf equipment. In this context every resource allocation must be justifiable as resulting from correct data, meaning that it is more important for the function points returned by an algorithm to be correct than it is for the algorithm to return all the correct function points.

These issues carry additional implications when designing programs for the public sector. The firm that produces programs for use by government agencies cannot overestimate the cost of a proposed project. Doing so can result in their bid being rejected as too expensive when the correct bid would have been accepted. Over-allocating resources to a government project can result in allegations of waste and mismanagement of funds, resulting in investigations where the contractor must account for every component of their estimate. Therefore it is even more important that cost estimates be based on correct function points rather than the estimate containing every correct function point.

It is possible that these results do not reflect how many of the relevant functions our approach returned in the context of programs produced by our industrial partner. The requirements for these programs are much more specific as they describe the exact operation of the program. The requirements use much of the same terminology for buttons and screen names that appears in the source code. We are unable to present a verbatim example from the industrial dataset as the requirements are

sensitive, but consider the following hypothetical requirement that contains a similar amount of detail and format:

> The "A" key shall invoke `do_add`. If `play_sounds` is set to true then it shall also send `PLAY_SOUND_ADD` to the asynchronous event service. If `logging_flag` is set to true then it shall send `KEY_A` followed by `CALL_DO_ADD` to the logging service.

Now consider this requirement taken from the JavaScript specification:

> 15.5.4.16 String.prototype.toLowerCase ( ) If this object is not already a string, it is converted to a string. The characters in that string are converted one by one to lower case. The result is a string value, not a String object.
>
> The characters are converted one by one. The result of each conversion is the original character, unless that character has a Unicode lowercase equivalent, in which case the lowercase equivalent is used instead.

The requirement is not as tightly bound to the source code as the previous hypothetical requirement. JavaScript requirements may contain pseudocode that specifies the exact behavior of a JavaScript library method but it does not require the implementation to use the same naming conventions as the pseudocode.

C. Limitations

We identified four limitations of our approach that could affect its accuracy or keep it from being easily generalized to all programming languages and environments. The first is that utility packages can inflate cost estimates. Utility packages contain code that is not relevant to the project domain. They tend to appear near the top of function point lists because they are called by all parts of the program. To ensure that utility packages are not inflating the number of function points, the user must either remove them from the call graph or exclude them from the returned list of function points.

Our implementation has no way of detecting deprecated code without either support from the language such as the "@depricated" annotation in Java or feedback from the user. This could present a problem if the function point list is used to inform developers that are implementing new features. A developer could inadvertently implement a change request using deprecated code since that was what appeared in the function point list. Alternatively, deprecated code could inflate the function point list if certain function points use new code and other function points use deprecated code. The function point list would then have multiple functions for the same task.

The algorithms we chose cannot detect indirect function calls such as those made through function pointers or anonymous functions. Indirect function calls were not part of the Ada standard used by the industrial partner, so we did not add support for them. Indirect function calls are common to programs with a plugin architecture where plugins register themselves by passing pointers to their component functions. The component functions are then called by the main program when necessary. The call graph builder must recognize that a library is being loaded from runtime and that certain functions are being requested. It must then find the appropriate source code that maps to that library and combine that call graph with the call graph of the main application.

Indirect function calls are also common to libraries that use callback functions. Examples include event handlers for GUI libraries such as Java Swing and entry points for new threads in libraries such as Pthreads. In these cases, a function pointer is passed to be executed by the library at some later time. Unless the source code of the library is available, the call graph builder cannot know exactly which function will call the function pointer; it could be the function to which it was passed or some other function. In cases where the source code to the library is unavailable, the user can assume that the function pointer will be called by the function to which it is passed. This assumption should not affect the complexity of the project and the resulting cost estimates. A call graph that does not contain indirect function calls will be incomplete. The function point list will also be incomplete and any PageRank scores will be invalid.

Finally, our implementation is unable to generate a call graph of projects that are made up of components written in multiple programming languages without extra effort. A builder must first be built to create a call graph for the new language, but it must also be built to recognize how the two languages interact. A simple case involves recognizing functions defined as extern "C" to allow C++ to call functions written in C. Another simple case involves calling assembly routines from C. Both of these cases allow the caller to simply address the function by name and the call graph is complete when the two call graphs are merged. More complex cases such as the Java Native Interface (JNI) load the binary version of the component at run time and call methods on an object that calls the target function inside the binary component. The call graph builder must first identify these calls and then must locate the callee in the call graph of the containing component. This way, the call graph is guaranteed to point to the appropriate function in cases where multiple components have a function with the same name.

Future work will focus on mitigating these risks. One area of future work is enabling the user to exclude deprecated and utility packages from function point analysis. We will first allow the user to manually select packages, but will eventually progress to suggesting packages to be excluded. Another area of work is the detection of indirect function calls where the particular language allows it. This will involve the detection not only of indirect function calls from within the program, but detection of function pointers being passed to outside libraries where it can be inferred that the outside library will then execute the function.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1313583. This work was also supported by the National Science Foundation CAREER Award under Grant No. CCF-1452959. This work was also supported by the United States Navy Small Business Innovation Research (SBIR) program under topic N141-055. Any opinions, findings, and conclusions expressed herein are the authors', and do not necessarily reflect those of the sponsors. The authors would like to thank SimVentions for their collaboration and for providing a subject matter expert for our user study.

REFERENCES

[1] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–82, 1994.

[2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[3] F. Heemstra, "Software cost estimation," *Information and Software Technology*, vol. 34, no. 10, pp. 627 – 639, 1992.

[4] T. Lee, T. Gu, and J. Baik, "Mnd-scemp: an empirical study of a software cost estimation modeling process in the defense domain," *Empirical Software Engineering*, vol. 19, no. 1, pp. 213–240, 2014.

[5] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: a software science validation," *Software Engineering, IEEE Transactions on*, no. 6, pp. 639–648, 1983.

[6] C. F. Kemerer, "Reliability of function points measurement: a field experiment," *Communications of the ACM*, vol. 36, no. 2, pp. 85–97, 1993.

[7] J. E. Matson, B. E. Barrett, and J. M. Mellichamp, "Software development cost estimation using function points," *Software Engineering, IEEE Transactions on*, vol. 20, no. 4, pp. 275–287, 1994.

[8] Y. Ahn, J. Suh, S. Kim, and H. Kim, "The software maintenance project effort estimation model based on function points," *Journal of Software maintenance and evolution: Research and practice*, vol. 15, no. 2, pp. 71–85, 2003.

[9] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the Seventh International Conference on World Wide Web 7*, ser. WWW7. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 1998, pp. 107–117. [Online]. Available: http://dl.acm.org/citation.cfm?id=297805.297827

[10] "(sbir) navy - automated function point analysis," http://www.navysbir.com/n14_1/N141-055.htm, accessed: 2014-09-18.

[11] K. Chen and V. Rajlich, "Case study of feature location using dependence graph." in *IWPC*. Citeseer, 2000, p. 241.

[12] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 11–20.

[13] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static techniques for concept location in object-oriented code," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. IEEE, 2005, pp. 33–42.

[14] N. Wilde and M. C. Scully, "Software reconnaissance: mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.

[15] A. D. Eisenberg and K. De Volder, "Dynamic feature traces: Finding features in unfamiliar code," in *21st ICSM'05*. IEEE, 2005, pp. 337–346.

[16] W. E. Wong, S. S. Gokhale, and J. R. Horgan, "Quantifying the closeness between program components and features," *Journal of Systems and Software*, vol. 54, no. 2, pp. 87–98, 2000.

[17] A. Marcus, A. Sergeyev, V. Rajlich, J. Maletic *et al.*, "An information retrieval approach to concept location in source code," in *11th WCRE, 2004*. IEEE, 2004, pp. 214–223.

[18] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JAsIs*, vol. 41, no. 6, pp. 391–407, 1990.

[19] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 351–360.

[20] S. Grant, J. R. Cordy, and D. Skillicorn, "Automated concept location using independent component analysis," in *15th WCRA, 2008*. IEEE, 2008, pp. 138–142.

[21] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[22] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[23] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. C. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *Software Engineering, IEEE Transactions on*, vol. 33, no. 6, pp. 420–432, 2007.

[24] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 234–243.

[25] M. Revelle, B. Dit, and D. Poshyvanyk, "Using data fusion and web mining to support feature location in software," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE, 2010, pp. 14–23.

[26] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empirical Software Engineering*, vol. 18, no. 2, pp. 277–309, 2013.

[27] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 604–632, 1999.

[28] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Configuring latent dirichlet allocation based feature location," *Empirical Software Engineering*, vol. 19, no. 3, pp. 465–500, 2014.

[29] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 522–531.

[30] A. Živkovič, I. Rozman, and M. Heričko, "Automated software size estimation based on function points using uml models," *Information and Software Technology*, vol. 47, no. 13, pp. 881–890, 2005.

[31] "OMG Automated Function Points, v1.0," Object Management Group, Tech. Rep., Jan. 2014. [Online]. Available: http://www.omg.org/spec/AFP/1.0/

[32] G. R. Finnie, G. E. Wittig, and J.-M. Desharnais, "A comparison of software effort estimation techniques: using function points with neural networks, case-based reasoning and regression models," *Journal of Systems and Software*, vol. 39, no. 3, pp. 281–289, 1997.

[33] G. Caldiera, G. Antoniol, R. Fiutem, and C. Lokan, "Definition and experimental evaluation of function points for object-oriented systems," in *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*. IEEE, 1998, pp. 167–178.

[34] S. Furey, "Why we should use function points [software metrics]," *Software, IEEE*, vol. 14, no. 2, pp. 28–30, 1997.

[35] C. Jones, *Estimating software costs: Bringing realism to estimating*. McGraw-Hill Companies New York, 2007.

[36] ——, "Backfiring: Converting lines of code to function points," *Computer*, vol. 28, no. 11, pp. 87–88, 1995.

[37] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 111–120.

[38] J. I. Maletic, M. L. Collard, and A. Marcus, "Source code files as structured documents," in *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. IEEE, 2002, pp. 289–292.

[39] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *Software Engineering, IEEE Transactions on*, no. 6, pp. 728–738, 1984.

[40] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[41] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of StatMT*, 2007, pp. 228–231.