

DYNAMIC MODEL GENERATION AND SEMANTIC SEARCH FOR OPEN SOURCE PROJECTS USING
BIG DATA ANALYTICS

A THESIS IN
Computer Science

Presented to the Faculty of the University
Of Missouri-Kansas City in partial fulfillment
Of the requirements for the degree

MASTER OF SCIENCE

By
SRAVANI PUNYAMURTHULA

B.Tech, Jawaharlal Nehru Technological University – Hyderabad, India, 2009

Kansas City, Missouri
2015

©2015

SRAVANI PUNYAMURTHULA

ALL RIGHTS RESERVED

DYNAMIC MODEL GENERATION AND SEMANTIC SEARCH FOR OPEN SOURCE PROJECTS USING BIG DATA ANALYTICS

Sravani Punyamurthula, Candidate for the Master of Science Degree

University of Missouri-Kansas City, 2015

ABSTRACT

Open source software is quite ubiquitous and caters to most common software needs developers come across. Many open source projects are considered better than their commercial equivalents as a larger pool of developers constantly improve it. However, one of the challenges to using open source is to manually analyze the code and understand the dependencies. Especially, for larger projects it is a very time consuming task. Hence, there is a strong demand for an automated process that could analyze the code and build an accurate model that represents the software system of the open source.

The objective of this thesis is to provide a solution to this problem by building a framework that can extract the features, identify components, connectors from the open source and provide the user a way to search functionality. The first step of this process is to extract the metadata and dependency information from the source code using a call graph. A call graph is a directed graph that represents the execution logic of the program and helps with analyzing the relationships between various classes. The extracted data is then transformed using Natural language processing (NLP) [15] techniques like lemmatization.

In the second step, the transformed data is semantically analyzed for feature extraction using Term Frequency Inverse Document Frequency (TF-IDF), synonym detection using Word2Vec [3] and component detection using Machine Learning dynamically. The dependency information extracted from the call graph is then used for identifying the connectors between the detected

components. Also, the dependency information is used to build a class dependency matrix that is further used for identifying dependency based components. In the final step, ontology is used to represent the features, components, connectors, classes discovered in the previous step and the relationships between them. The generated ontology can be queried to search for functionality using the SPARQL [5] query language. Protégé [4] is used for visualization of the generated ontology.

The proposed solution is built on Spark, a parallel processing framework and provides a fully automated and scalable model for representing the software. In this thesis, we have analyzed two open source projects Apache Solr and Apache Lucene as a case study. Apache Solr is built using Apache Lucene core library. The results from Apache Solr analysis are compared to the manual evaluation of software architecture by experts. We have observed that 90% of the features identified in the manual analysis are recovered in the automated approach and also many new features are discovered. This thesis also analyzes the dependencies between the components detected for Apache Solr and Apache Lucene projects. From this analysis of the two systems, we have observed that Apache Solr is highly dependent on Apache Lucene.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis titled “Dynamic Model Generation and Semantic Search for Open Source Projects using Big Data Analytics” presented by Sravani Punyamurthula, candidate for the Master of Science degree, and hereby certify that in their opinion, it is worthy of acceptance.

Supervisory Committee

Yugyung Lee, Ph.D., Committee Co-Chair
School of Computing and Engineering

Yongjie Zheng, Ph.D., Committee Co-Chair
School of Computing and Engineering

Praveen Rao, Ph.D.
School of Computing and Engineering

TABLE OF CONTENTS

ABSTRACT	iii
ILLUSTRATIONS.....	viii
TABLES	x
Chapter	
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Proposed Solution	3
2. BACKGROUND AND RELATED WORK	4
2.1 Terminology.....	4
2.2 Related Work.....	5
3. PROPOSED FRAMEWORK.....	8
3.1 Overview.....	8
3.2 Data Extraction	10
3.2.1 Call graph.....	12
3.2.2 Class Document Creation.....	15
3.2.3 Class Dependency Matrix.....	16
3.2.4 Extracting Interfaces.....	17
3.3 Data Transformation.....	18
3.3.1 Camel Case Split.....	18
3.3.2 Lemmatization.....	19
3.4 Feature Based Component Detection.....	20
3.4.1 Word Count.....	20
3.4.2 Word2Vec.....	21
3.4.3 Classification.....	23
3.4.4 Component Detection.....	25

3.4.5 Connector Identification.....	26
3.5 Dependency Based Component Detection.....	27
3.6 Component Specific Feature Extraction.....	28
3.7 Ontology and Query Processing.....	31
4. RESULTS AND EVALUATION.....	36
4.1 Apache Solr.....	36
4.1.1 Feature Based Analysis	39
4.1.2 Dependency Based Analysis	46
4.1.3 Evaluation	48
4.1.4 Ontology and Query Processing	53
4.1.5 Performance Analysis.....	62
4.2 Apache Lucene	65
4.2.1 Feature Based Analysis	68
4.2.2 Dependency Based Analysis	74
4.2.3 Evaluation	77
4.2.4 Performance Analysis.....	79
4.3 Comparison between Apache Solr and Lucene.....	80
5. CONCLUSION AND FUTURE WORK.....	84
5.1 Conclusion	84
5.2 Limitation.....	85
5.3 Future Work	85
REFERENCES.....	86
VITA.....	88

ILLUSTRATIONS

Figure	Page
1: Workflow of Proposed Solution	9
2: Implementation Platform.....	10
3: Data Extraction	11
4: Data Extraction Complete Flow.....	11
5: Class Document Creation	15
6: Degree of Dependency Computation.....	17
7: Data Transformation Example.....	18
8: Camel Case Split Example.....	19
9: Feature Based Component Detection	20
10: Word Count	21
11: Word2Vec Example	22
12: Supervised Learning	23
13: Decision Tree Example	24
14: Classification Example	25
15: Connector Example	27
16: Component Specific Feature Extraction.....	29
17: Feature Extraction Example	31
18: Software Ontology	33
19: Ontology Generation	34
20: Class Only vs Class + Dependency Input Data	39
21: Apache Solr Important Words	40
22: Sample Data for Clustering	46

23: SSE Plot for Different Cluster Sizes.....	47
24: Component Evaluation with Manual Analysis.....	49
25: Component Evaluation with Package Clusters.....	50
26: Feature Evaluation with Manual Analysis	51
27: New Features discovered using Proposed Model	53
28: Components generated using Different Approaches.....	54
29: Component to Class Visualization	55
30: Ontology - Features.....	56
31: Feature - Feature Variant - Component Map.....	56
32: Class to Feature Variant Visualization	57
33: Component to Connector Visualization	58
34: OntoGraf - View Details	59
35: OntoGraf - Search Functionality	60
36: SPARQL Query to Fetch Features	61
37: SPARQL Query to Fetch Variants of a Feature	62
38: Run time Statistics for Machine Learning Algorithms	64
39: Apache Lucene Important Words	68
40: Sample Dependency Vectors for Apache Lucene.....	75
41: Component Evaluation with Packages for Apache Lucene	78

TABLES

Table	Page
1: Callgraph Types of Call.....	13
2: Callgraph Example	14
3: Entity Weightage	16
4: Word Count Example.....	21
5: Feature Based Component Example	26
6: Apache Solr Statistics.....	37
7: Sample Callgraph Output.....	37
8: Sample Callgraph Statistics.....	38
9: Naïve Bayes Classification Output for Class + Dependency Data.....	41
10: Naïve Bayes Classification Output for Class Only Data.....	41
11: Feature Based Components for Class + Dependency Data.....	42
12: Feature Based Components for Class Only Data.....	43
13: Sample Component to Feature Mapping.....	44
14: Sample Class to Feature Variant Mapping	45
15: Sample Connectors in Solr.....	45
16: Sample Dependency Based Components.....	47
17: Sample Components using Manual Analysis Data.....	48
18: Partial Matching Results of Features.....	52
19: Solr Intermediate Outputs.....	63
20: Run Time Statistics.....	63
21: Apache Lucene Statistics.....	65

22: Apache Lucene Sample Call graph Output.....	66
23: Apache Lucene Sample Callgraph Statistics	67
24: Naïve Bayes Sample Output for Apache Lucene Class only data.....	69
25: Feature based Components using Class only data.....	70
26: Naïve Bayes Sample Output for Apache Lucene Class+Dependency Data.....	71
27: Feature Based Components Sample Output using Class +Dependency Data.....	72
28: Apache Lucene Sample Component to Feature Mapping	73
29: Sample Class to Variant Mapping	73
30: Sample Connectors in Apache Lucene	74
31: Sample Dependency Components for Apache Lucene.....	76
32: Sample Output for Component to Package Match	77
33: Lucene Intermediate Outputs.....	79
34: Run time Statistics of Apache Lucene.....	80
35: Apache Solr and Lucene Comparison	80
36: Common Features between Solr and Lucene.....	81
37: Sample Dependency Data between Apache Solr and Lucene	82
38: Sample Component Dependency between Solr and Lucene	83

CHAPTER 1

INTRODUCTION

1.1 Motivation

Open source software is quite ubiquitous and caters to most common software needs developers come across. Many open source projects are considered better than their commercial equivalents as a larger pool of developers constantly improve it. However, one of the challenges to using open source is to manually analyze the code and understand the dependencies. Especially, for larger projects it is a very time consuming task. Software developers come across similar problems often, hence there is a high possibility that a part of the code for the application is already available online. Reusing the software code not only reduces the time lines for software development but also enhances the design of the software application.

There are vast open source repositories available online. Significant amount of code which is being written today has already been written before and is available through open source repositories like GitHub. Unfortunately, very little open source code is being reused because of the undocumented evolution of software systems and the complex nature of architecture which is very difficult to be analyzed manually. Also, as the number of open source repositories increase, finding the relevant source code becomes difficult. GitHub [1] and Source Forge [2] are some of the famous open source repositories which host more than 20 million repositories. Also, the existing open source repositories like GitHub provide only keyword based search that may not retrieve relevant projects. Once the required source code is retrieved, a programmer needs to analyze the software system. This is a very time consuming task to analyze the structure and architecture of the software project from the source code. Since open source projects evolve continuously, it is very difficult to rely on manual analysis for understanding the software.

Sometimes the code is too complex to understand and may contain hundreds of files or classes located in different packages and the user is required to navigate back and forth between pieces of code to understand the true nature of interaction between classes. The manual analysis although being very time consuming and tedious does not guarantee accuracy and fails to capture the exact nature of interaction between classes and hence is not capable of presenting an accurate picture of the architecture. All these factors have been discouraging the programmers from reusing this high quality open source code which drives the need for an automatic architecture recovery model. The approach for architecture recovery presented in this research makes use of big data analytics and machine learning algorithms to automatically generate a highly accurate model of a software system. This approach can also be used to compare software systems based on design and architecture. This approach also finds application in comparing the various versions of projects like Lucene, Solr, and Hadoop in terms of software systems and detect the newly incorporated features.

1.2 Problem Statement

Software programmers can reuse the high quality open source code in their applications. However, open source projects are huge and complex in nature. Hence manually analyzing the code is very time consuming. There are solutions that help in understanding the software but the existing solution provide very little information about the functionality and dependencies in the given software project. Most of the open source projects have been written using Object Oriented programming style which means its semantics, structure and dependencies can be exploited to retrieve most of the architecture elements of the project. Since open source projects tend to be huge, the solution must be scalable and cost effective. The application should be user friendly that allows users to query the recovered model to search for functionality.

1.3 Proposed Solution

This thesis focuses on analyzing an open source project and automatically building a model that represents the project accurately. In order to build a model automatically, the following architecture elements are extracted from the project using machine learning: features, components, connectors, component to feature mapping, class to feature variant mapping.

The proposed solution extracts the calling relationships available in the project using call graph. It performs two types of analysis on the extracted data: semantic analysis and dependency analysis. For the semantic analysis, the required metadata about a class (e.g., package name, class name, method names, parameter names, calling class's names, calling method names, etc.) are extracted from the call graph output and a class document is created. The metadata is then transformed using Natural language processing techniques like lemmatization to convert the words to their base forms and remove stop words and special characters. The transformed data is then analyzed using machine learning algorithms like Word2Vec [3] and Naïve Bayes to identify the feature based components. For the dependency analysis, the dependency information extracted from the call graph output is used to generate a class dependency matrix which is passed as an input to K-Means [3] clustering algorithm. K-Means [3] algorithm clusters the classes into dependency based components. Once the components are detected using either of the two approaches, the features specific to the components is extracted using Term Frequency Inverse Document Frequency (TF-IDF) [1].

The approach presented makes use of big data analytics and machine learning techniques to automatically generate a highly accurate model of a Software System's Architecture. The generated model can be visualized using tools like Protégé [4] and queried using SPARQL [5] query language. The proposed solution is built on Spark that is a parallel processing framework. The scalability and performance of the proposed approach is tested on a Spark cluster. Also, the results obtained from the proposed solution are evaluated against the results from the manual analysis.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Terminology

In this chapter, the key terms related to software architecture are introduced and their interpretations in the context of this paper are defined.

Software Architecture represents the higher level structure and design of a software system similar to a blueprint. Software architecture is defined in terms of components and the interrelationship between them. In software, a code entity is a programming snippet. In a Java project, sample code entities are Class, Package, Interface, Method etc.

A *Software Feature* is commonly defined as a characteristic of the system that is visible to the end user. A feature can also be a functionality, a configuration or a software design decision. According to Feature Oriented Domain Analysis (FODA) [6], features can have sub features. Features of a software project can be categorized as mandatory, optional and alternative. In this thesis, feature is basically a word representing a functionality implemented by a class. Also, we have not classified the features as optional, mandatory, and alternative.

A software *Component* is a module that encapsulates related functions and data. A component can be a package, web service, rest API etc. Software components interact with each other through interfaces. Component based architecture is mainly used to improve reusability and reduce dependency. In this thesis, we have defined two types of components: feature based and dependency based. Feature based component is a group of classes that implement similar functionality. Dependency based component is a group of classes that are highly related to each other. There can be many-to-many relationships between features and components as a feature may be implemented by multiple components and a component may provide multiple features depending on the software design.

Connectors are used to analyze the component interactions and dependencies in the software project. In this thesis, a connector is defined as the directed dependency relationship between the components.

Call graph is a directed graph that represents the overall execution of the program and helps in analyzing the calling relationships between different classes. There are two types of call graph: static and dynamic. A static call graph represents every possible run of a program. Whereas a dynamic call graph is more specific and represents only one execution of the program. In this thesis, static call graph information is used for analyzing the features and components in the given project.

2.2 Related Work

In general, software architecture recovery requires a design expertise and domain knowledge. This is a knowledge based approach in which a software engineers with domain expertise try to reverse engineer the source code to recover the software architecture. This approaches may work well for smaller projects with less complexity. However, analyzing larger projects takes a lot of time and also does not guarantee accuracy. Hence there is a need for alternative solutions that are fully or semi-automated.

Belady and Evangelisti [7] were the first to cluster the software system based on the information extracted from the systems documentation. This approach depends on the documentation and does not analyze the source code. The major disadvantage with this approach is that the software documentation may not be up to date. Anquetil and Lethbridge [8] have extracted the names of the entities for understanding the functionality of the program. Garcia and Mattmann [9] proposed establishing prototypes of architectures using a recovery scheme that relies on the domain expertise of system engineers. In our approach, there is no need for manual intervention as the metadata required for analysis is automatically generated from the source code and then machine learning algorithms are applied on the generated metadata for analysis.

Paydar and Kahani [10] proposed a semantic web based approach for detecting the design patterns from the source code. DP-Miner [11] proposed a similar approach to discover design patterns using matrix representation of structural characteristics of the software system. Sartipi-K [12] proposed an approach to recover software architecture based on pattern matching. In this approach a software project is represented as a graph by using data mining techniques that extract highly related code entities. Then it requires a software engineer with domain knowledge to create an architectural query that represents the architectural pattern of the system which is then represented as a graph. The pattern graph is then matched with the system graph to recover the software architecture of the system. There are two major drawbacks with these approaches. Firstly, these approach require an active participation of software engineers and secondly, there approaches cater to only design pattern detection. Also, most of the logic is hard coded for a specific system, hence there is a need for more generalized solution.

In the proposed model, all the metadata related to a class is extracted to prepare class documents and two types of components are generated based on the similarity in features and dependencies. While generating the metadata, weightages are assigned to the calling relationships based on Object Oriented principles and further analysis is performed to determine their behavior. All the steps required for analysis like the data extraction, feature discovery, component detection etc are generalized for any software project written in Java. Hence the proposed model builds a completely automated solution for analyzing a software project and building a model to represent the same.

RDFCoder [13] is a library that parses a Java project and analyzes the package structures, classes, interfaces, libraries, and the relationships between them. This library generates a model of the given software project similar to our approach. RDFCoder [13] uses the Kabbalah model for representing the code entities and the relationships between them in Resource Description Framework (RDF) [14] format. RDF [14] is a standard format for data interchange on the web. The major limitation

with RDFCoder [13] is that it captures only the structure of the code and package to class hierarchy. The proposed solution discovers the features, feature variants and detects the components based on similarity in features and dependency relationships. It also creates a feature to component, component to class and class to feature variant mappings and generates an ontology to represent these relationships.

CHAPTER 3

PROPOSED FRAMEWORK

3.1 Overview

There are two main approaches implemented for component detection and feature extraction of the open source java projects, unsupervised and supervised learning. Two kinds of metadata are extracted for analysis, semantic information and dependency information. Figure 1 shows the overall architecture of the proposed framework. This application is focused on analyzing java projects, so the input to the application is a Java source jar. The architecture of the application consists of six key modules, data extraction, data transformation, feature based component detection, and dependency based component detection, feature extraction, visualization and query processing.

The data extraction component takes care of Metadata generation. The input to the preprocessor is the java source jar. The application generates two different types of Metadata, one based on semantics like package names, class names, method names, attribute names and the other from the calling relationships between the classes. In this step, the metadata is processed to generate class documents and class dependency matrix. All the intermediate documents generated in this approach are stored in the Hadoop distributed file system.

The class documents generated in the data extraction step are then transformed using Natural language processing (NLP) [15] techniques like lemmatization and stemming. The transformed data is then sent to the feature based component detection module. In this module, word count program is used to analyze the important features and Word2Vec [3] machine learning algorithm is used to build the feature synonym vectors which form the training data for the supervised learning algorithms. Machine learning algorithms like Naïve Bayes classifier, Random Forest are used for supervised learning. Once the model is trained using feature synonym vectors, the class metadata documents are

used to identify the key feature in each class. The classes are then grouped to form components based on the features identified in supervised learning. The components identified in this module are then passed to the component specific feature extraction module. In this module, TF-IDF [3] is used to identify the key features in each component.

The class dependency matrix created in the data extraction module is used to build the class dependency vectors. These vectors are used as an input to the K-Means [3] clustering algorithm. K-Means [3] is an unsupervised learning algorithm. It clusters the classes based on the similarities in the dependency relationships. These clusters form the dependency based components for the given project.

The information extracted in the different modules like the features, components, component to class mapping, component to feature mapping, feature to feature variant mapping, class to feature variant mapping etc. are then used to build ontology which can be visualized using Protégé [4] tool and queried using SPARQL [5] query language.

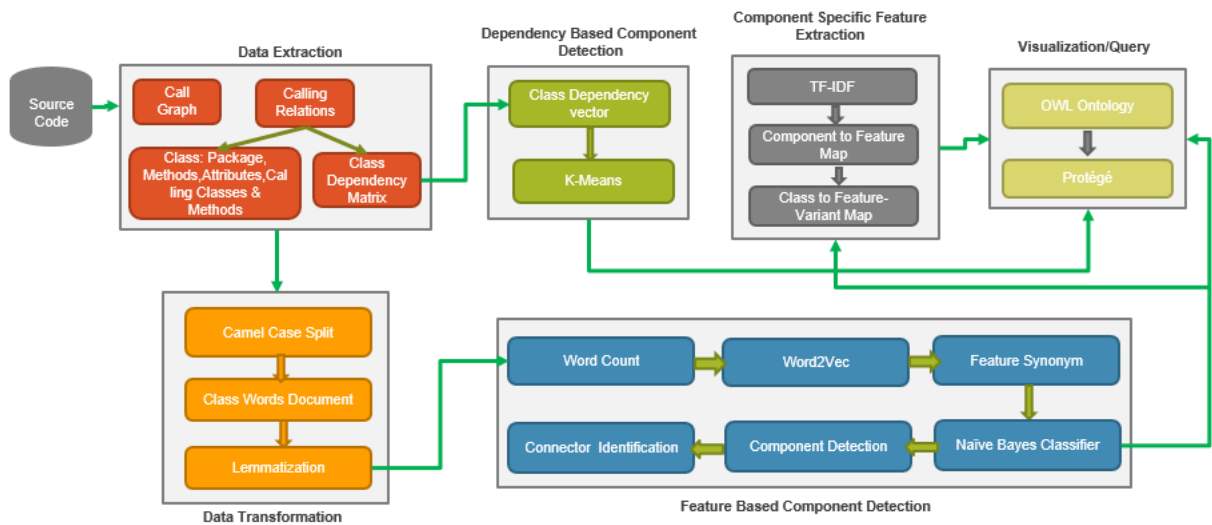


Figure 1: Workflow of Proposed Solution

The proposed solution is built on Spark environment. Apache Spark is an open source parallel processing framework. Apache spark provides performance up to 100 times faster compared to Hadoop

due to its in-memory storage mechanism. In Spark, data from the software programs can be loaded into the memory of the cluster machines, thus providing faster retrieval of the data. Hence Spark is most suitable for machine learning.

Apache Spark provides Mllib [3] that is a scalable machine learning library. It consists of common machine learning algorithms for classification, regression, clustering, etc. Scala and Java programming languages are used for implementing the program logic. Protégé [4] tool is used for visualization of the ontology generated as part of the thesis.

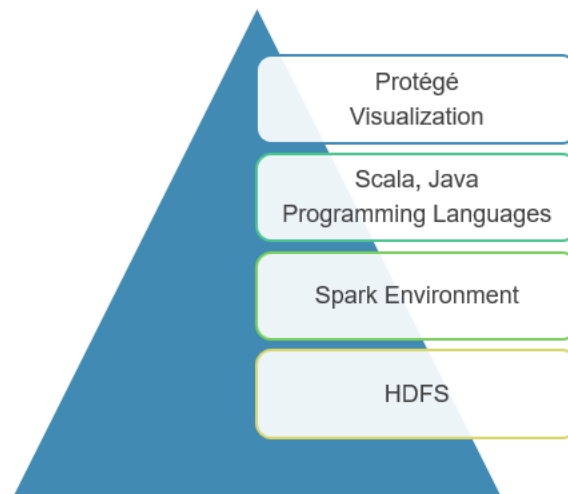


Figure 2: Implementation Platform

3.2 Data Extraction

Data extraction is the first module of the proposed model. This module is used to extract all the required information from the source code. Since, we have developed this model for Java projects, the input to this module is a source jar. This module consists of four steps: extracting calling relationships using call graph, generating class metadata document, generating class dependency matrix, extracting interfaces.



Figure 3: Data Extraction

The following figure shows the complete flow of the data extraction module. In the first step, Java call graph open source project is used to extract the calling relationships from the given source jar. The output of this call graph is then used in the second step to extract the required metadata like the class name, package name, method names, calling classes, calling methods etc. In the third step, the call graph output is used to generate a class dependency matrix by computing the degree of dependency between the classes. Each of the steps implemented in this module are explained in detail in the later sections.

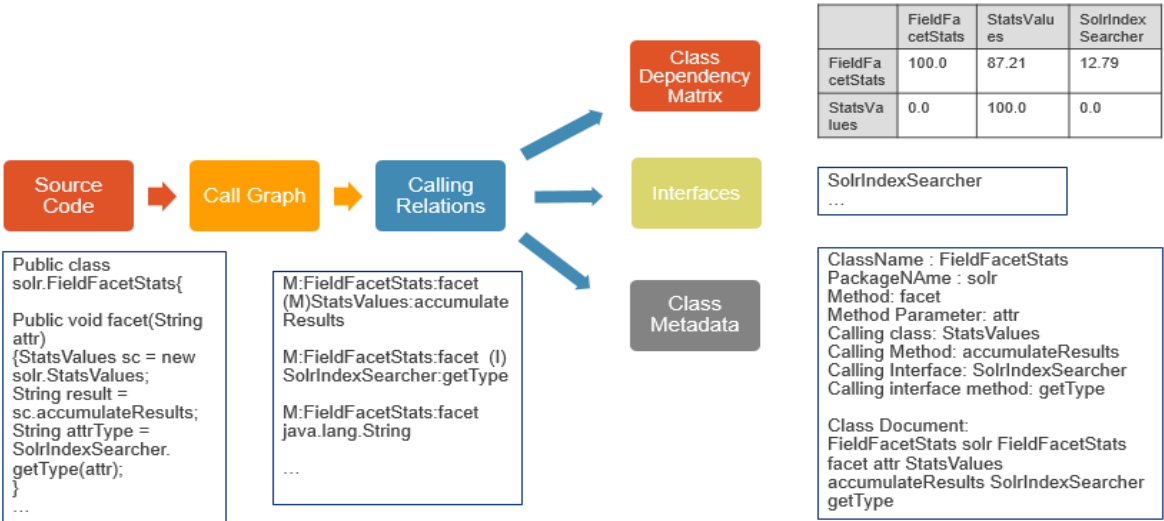


Figure 4: Data Extraction Complete Flow

3.2.1 Call Graph

Call graph is a directed graph that represents the overall execution of the program and helps in analyzing the calling relationships between different classes. There are two types of call graph: static and dynamic. A static call graph represents every possible run of a program. Whereas a dynamic call graph is more specific and represents only one execution of the program. In this thesis, static call graph information is used for analyzing the features and components in the given project.

Java Call graph [16] is an open source project developed by the Regents of the University of California. We have used this project to extract all the calling relationships available in the source code. Java Callgraph Master [16] is a suite of programs used to generate both static and dynamic call graphs. It can be used both as a library or a command line tool. We have used static call graphs to analyze the calling relationships between various classes in the source code. The call graph program reads byte code of the classes from a jar file, walks down the method bodies and generates a table of caller-caller relationships in the following format:

M:class1:method1 (type of call)class2:method2

This line means that method1 of class1 called method2 of class2. There are four types of calls based on the java opcode: invoke virtual, invoke interface, invoke special, invoke static. The type of call depends on whether the callee is a class or interface or super class etc. For example, if a class calls a method in its super class then it is represented by the opcode invoke special. Similarly, if a class calls a method in an interface, then it is represented by invoke interface. The table 1 below explains the types of calls with examples.

Table 1: Call graph Types of Call

Type of Call	OpCode Instructions	Example
M for invokevirtual calls	Used to invoke instance methods	M:org.apache.solr.HttpSolrServer:request (M)org.apache.solr.SolrRequest:getMethod
I for invokeinterface calls	Used to invoke methods of interfaces	M:org.apache.solr.HttpSolrServer:request (I)java.util.Collection:iterator
O for invokespecial calls	Used to invoke instance initialization methods as well as private methods and methods of a superclass of the current class.	M:org.apache.solr.HttpSolrServer:request (O)org.apache.solr.SolrServerException:<init>
S for invokestatic calls	Used to call the class methods that are declared with the static keyword	M:org.apache.solr.HttpSolrServer:request (S)org.apache.solr.util.ClientUtils:toQueryString

Following example explains how the calling relationships are extracted using the opcode in java. The example has 2 classes scheduler, JobImp and an interface Job. The interface Job has only one method: execute. JobImp is a class that implements the interface Job. Scheduler class has two static methods: main, print. The main method in the scheduler class calls the execute method of the interface job and then prints the result using the print static method. The call to the execute method is given the opcode invoke interface and the call to the static method print is given the opcode invoke static in the bytecode. Also, every class in Java is a subclass of the Java.lang.object class, hence it is represented with the opcode invoke special. The below table 2 displays the sample code, the byte code and the output of the call graph.

Table 2: Call graph Example

Source Code	OpCode for Scheduler Class	Callgraph Output for Scheduler class
public class Scheduler {Job	public class Scheduler extends	M:Scheduler:<init>
job = new JobImpl();	java.lang.Object{	(O)java.lang.Object:<init>
public static void main() {		
String result = (String)	1:invokepecial #1;	M:Scheduler:main
job.execute(); print(result);	//Method	(I)Job:execute
}	java/lang/Object."<init>":()	
private static void print(String		M:Scheduler:main (S)
message){	public void main();	Scheduler:print\$message
System.out.println(message);	4: invokeinterface #5,	
}}	//InterfaceMethod	M:Scheduler:print\$message
	Job.execute():Ljava/lang/Object	(M)java.io.printStream:print
public interface Job { Object	14: invokestatic #7;	In
execute(); }	//Method print:(java/lang/String;)	
	private static void	
public class JobImp	print(java.lang.String);	
implements Job { public	4: invokevirtual #9;	
Object execute(){	//Method	
Integer value =	java/io/PrintStream.println:(Ljava/la	
createRandomValue();	ng/String;)	
return incValue(value);		

3.2.2 Class Document Creation

The output generated from call graph is parsed to extract the required information about the class like the package name, class name, method names, parameters and the calling class names and method names. The metadata extracted about each class is used to create a class metadata document. The class metadata document has the class name followed by all the related words delimited by space.

Format of the class document: `ClassName <related words>`

Two types of metadata documents are generated for a class: class only, class + dependency. The class only document consists of information about the class like the package name, class name, methods, attributes. The class + dependency also consists of the calling class names and calling method names. While creating the class document, only a substring of package name is included. For example, if the package name is `org.apache.solr.search`, then only the term `search` is added to the class document because the portion `org.apache.solr` is common in all the packages, hence it does not add any value for semantic analysis.

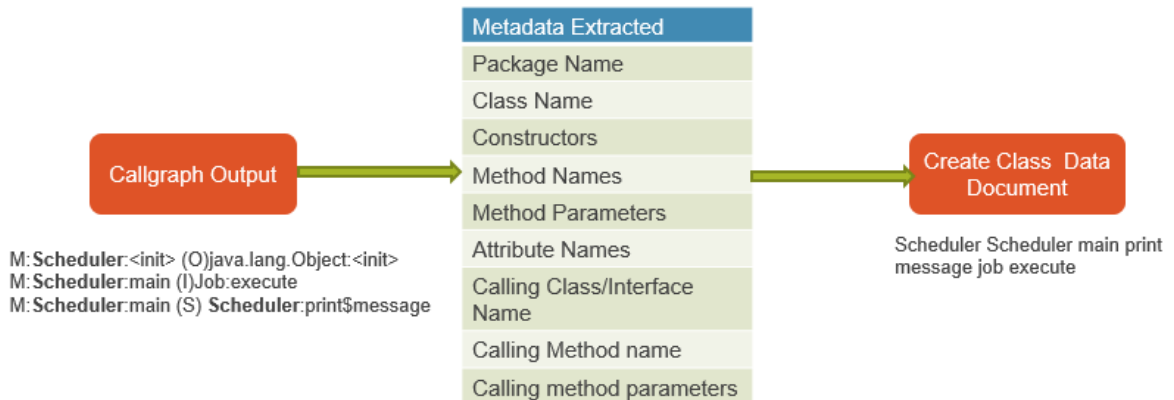


Figure 5: Class Document Creation

3.2.3 Class Dependency Matrix

The call graph output is used to analyze the degree of dependency between the classes. The call graph data presents different types of calling relationships between the classes. The type of dependency determines the closeness between the classes. For example, invoke special is used to call the methods in the super class. Hence it captures the abstraction relation between the classes. Since the classes are closely related to their super classes, the super classes should be given more weightage while computing the degree of dependency. Also, generally a package is a way of grouping related classes. Hence classes within the same package is given more weightage. Similarly, all the calling types are analyzed and a weightage for each type is determined. The following table 3 shows the weightages given to different calling types.

Table 3: Entity Weightage

Resource	Weightage
Package	20
InvokeSpecial	15
InvokeVirtual	10
InvokeStatic	10
invokeInterface	5

There are three steps for determining the degree of dependency: calculate count, calculate weight, and calculate degree. Firstly, we need to calculate the count that is the number of times a class is being called by the given class. Once the count is determined, we need to identify the type of call and multiply the count with the weightage to obtain the weight. Finally, the degree of dependency on a class is obtained by dividing the class weight with the total weight. The calculations can be represented mathematically as follows.

$$\text{Weight} = \text{Count} * \text{Weightage}$$

$$Total\ Weight = \sum count * weightage$$

$$Degree = \left(\frac{count * weightage}{total\ weight} * 100 \right)$$

The following example shows the flows of computation of degree of dependency for a sample class:

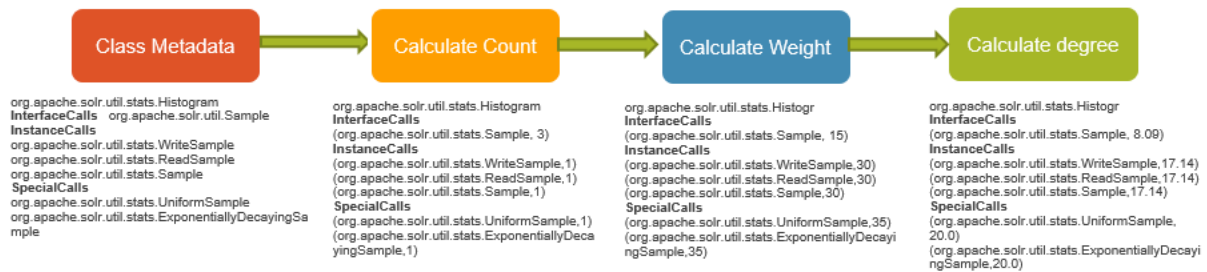


Figure 6: Degree of Dependency Computation

3.2.4 Extracting Interfaces

Call graph identifies the type of calling relationship between the classes. One of the type of calls is invoke interface. This relationship is identified with the letter I. For example, the below output shows that the load method in ManagedResourceStorage is calling the info method in logger interface.

M:org.apache.solr.rest.ManagedResourceStorage:load (I)org.slf4j.Logger:info

From this output, we can extract the interface org.slf4j.Logger. Similarly, all the interfaces present in the source code are extracted. The proposed model could be extended in future to identify the components that implement each of these interfaces extracted and also to classify the interfaces as internal and external interfaces.

3.3 Data Transformation

Data transformation is the second module in the proposed model. In this module, the class metadata documents generated in the data extraction module are transformed into a format suitable for analysis. This module has two main steps: camel-Case Split, lemmatization. The output of this module is class words documents. The format of the class words document is given by:

ClassName (\t) relatedWords(delimited by space)

The complete flow of data transformation is explained using an example in the following figure. The details of each of the steps are explained in the following sections.

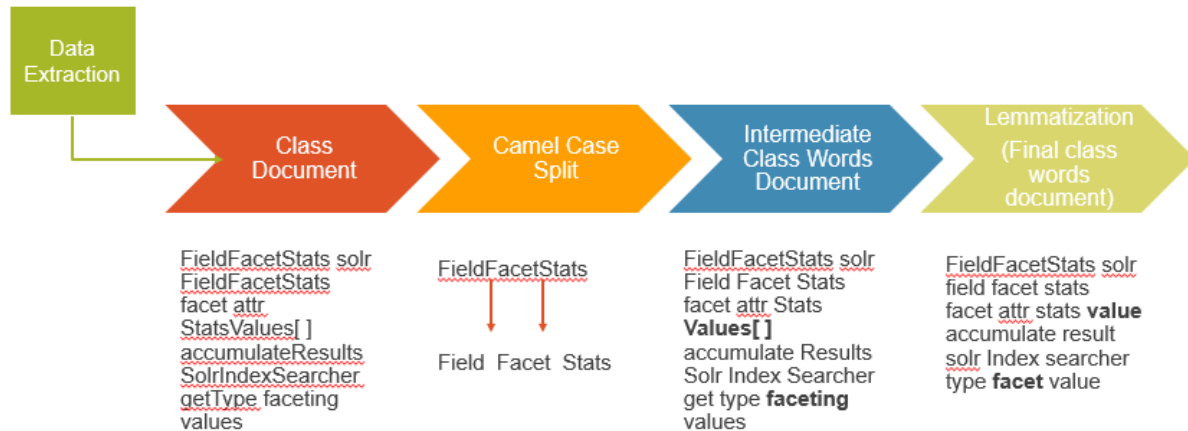


Figure 7: Data Transformation Example

3.3.1 Camel Case Split

Camel case is a naming convention commonly used in Java. In camel casing two or more words are combined together such that each word begins with a capital letter. So the words need to be split based on this naming convention in order to identify the individual words for semantic analysis. The input to this step is the class metadata document generated in the data extraction module. Each word in the metadata document is split based on camel case naming convention and a class words document

is generated. In the class words document, each line holds a class name followed by all the words delimited by space.



Figure 8: Camel Case Split Example

3.3.2 Lemmatization

Lemmatization is a Natural language processing technique. Stanford CoreNLP [15] is used as a library for implementing lemmatization. Stanford CoreNLP is an integrated framework that provides a sets of natural language processing tools for linguistic analysis. Lemmatization basically converts a word to its base form with the use of vocabulary. The base form of a word is known as the lemma. For example: “clients” and “client’s” are converted to “client”. Lemmatization is very crucial for semantic analysis, otherwise the words clients and client are identified as different features. Since the model aims to group the classes to components based on similar features, it is crucial to lemmatize the words to improve accuracy.

In this step, the following transformations are applied to the class words document:

- 1) Converts a word to its base form
- 2) Removes stop words
- 3) Removes special characters like \$, &, [] etc.
- 4) Converts words to lower case.

Stop words are the most common words such as “the”, “is”, “at”, “for” which does not add any value to the semantic analysis for identifying important features. Hence the stop words need to be filtered before processing the data.

3.4 Feature Based Component Detection

This module finds the components based on the similarity of features in the classes using supervised learning approach. The input to this module is the class words document generated in the data transformation module. The input is processed to identify the important features using word count program, then Word2Vec [3] machine learning algorithm is used to identify the synonyms for these features. Supervised learning algorithms like Naïve Bayes, Random forest are used for classify the classes based on the features. The process in this module can be divided into five steps as shown below:



Figure 9: Feature Based Component Detection Steps

3.4.1 Word Count

Word count program is used to identify the important features. The class words document from the data transformation module is used as an input to this program. The program computes the word count for each class and the words are sorted in the decreasing order of frequency. The top two words for each class are taken and grouped together to identify the key features implemented in the project.

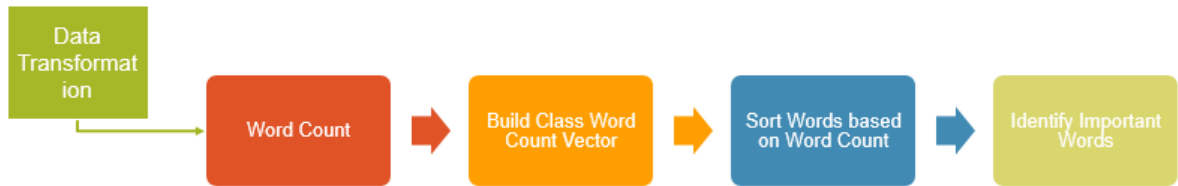


Figure 10: Word Count

The below example shows a sample class and the corresponding word count vector. Each class is identified by a unique number. Similarly, each word is identified by a unique number. Two words are picked from each class. The output of this program is a list of important words. The important words for this example are stats, value.

Table 4: Word Count Example

Sample Class	Word Count Vector
org.apache.solr.util.stats.Histogram	(500,
histogram stats sample value clone	([0,1,3,7,19,24,47,296,350],
value clinit clear update sample stats stats clear	[1.0,3.0,2.0,3.0,1.0,1.0,2.0,2.0,1.0]))
update snapshot value	

3.4.2 Word2Vec

Word2Vec [3] is a neural network language model which takes sequences of words that represents documents and computes distributed vector representation of words. Spark Word2Vec [3] uses skip gram model for representing the words. In skip gram model, each word is associated with 2 vectors say uw and vw which are vector representations of the w as word and context. The main advantage of distributed vector representation is that similar words are closer in the vector space.

In our model, we are using Word2Vec [3] to extract the synonyms for the important words identified using the word count program. The synonyms are detected based on the cosine distance between the vectors representing the words. The similarity or the cosine distance between the vectors is computed using the below formula.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}}$$

The output of the Word2Vec [3] program is a feature synonym vector. For each of the important words, identified in word count program, the synonyms are detected and a feature synonym vector is constructed using the cosine similarity.

In this example, the input to the word to vector program is the “stats” word. Word2Vec [3] analyzes all the words in the class documents and identifies the synonyms for the given word.

Synonyms →

The synonyms of the word and their cosine distances are used to create the feature synonym vector as shown in the figure 10.

Feature Synonym Vector →

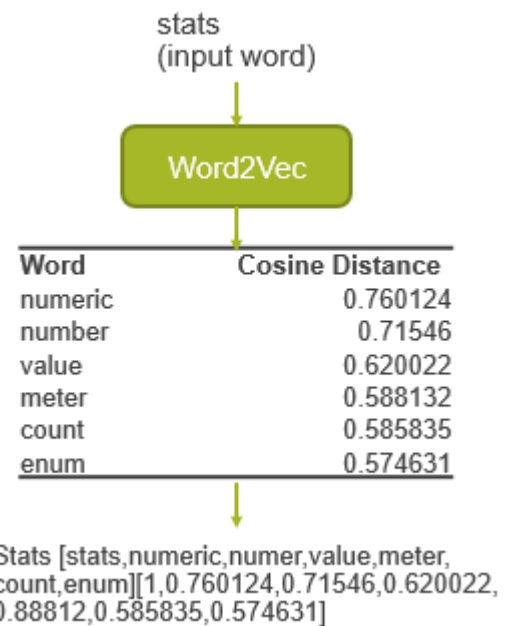


Figure 11: Word2Vec [3] Example

3.4.3 Classification

Supervised Learning algorithms are used for classification of classes based on the features. The following figure demonstrates the steps involved in supervised learning:

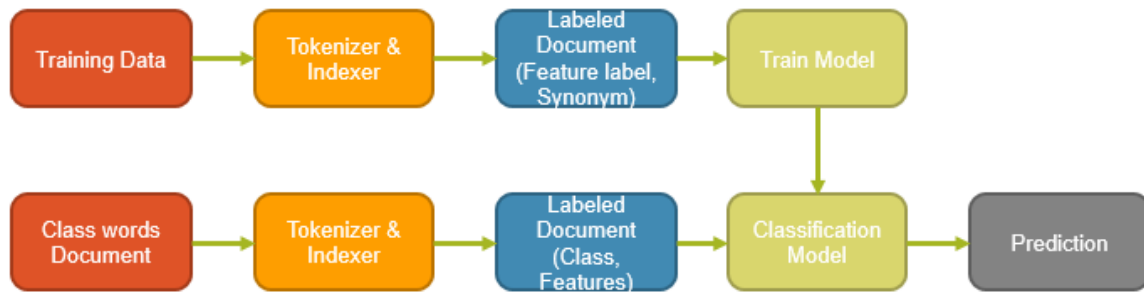


Figure 12: Supervised Learning

There are two types of supervised learning algorithms: classification and regression. In our model, we are using the classification algorithms. The classification algorithms classify the given data based on the labeled training data. The feature synonym vectors created using the Word2Vec [3] algorithm form the training data for the classification algorithm. The label is the feature and the synonyms form the data for training. The feature synonym vectors are converted to labeled document and then used for training the model.

The class words documents are then converted to labeled vectors and used for testing the model. The model predicts the key feature in each class document based on the training data. Two classification algorithms are implemented for this purpose: Naïve Bayes classifier and Random Forest. The accuracies of both the algorithms are tested.

Naive Bayes is a multiclass classification algorithm that relies on conditional probability. During training, the Naïve Bayes algorithm computes the conditional probability of the data given the label. Once the model is generated using the training data, Naïve Bayes computes the conditional probability of the label for the given testing data and predicts the label with highest probability.

The labeled documents obtained from the feature synonym vectors are used for training the model. Naïve Bayes computes the conditional probability of each synonym given the feature using the following formula:

$$P(\text{Word}|\text{Label}) = \frac{P(\text{Word} \cap \text{Label})}{P(\text{Label})}$$

The class words documents are used for classifying the classes. During prediction, the algorithm applies Bayes theorem to calculate the conditional probability of the label given the words in the class document and predicts the label with the highest probability.

$$P(\text{Label}|\text{Word}) = P(\text{Word}/\text{Label})P(\text{Word} \cap \text{Label})/P(\text{Word})$$

Random Forest [3] is an ensemble of decision trees. The decision tree [3] is a supervised learning classification algorithm that takes a greedy approach to recursively partition the feature space. The tree predicts the same label for all the bottommost leaves. Each partition is chosen by selecting the best split from a set of possible splits to maximize the information gain at a tree node. Random forest [3] creates multiple decision trees in parallel. It incorporates randomness by sampling. For each decision tree, it takes a random subset of training data and a random subset of features. During prediction, random forest aggregates the data from all the decision trees and considers the majority vote.

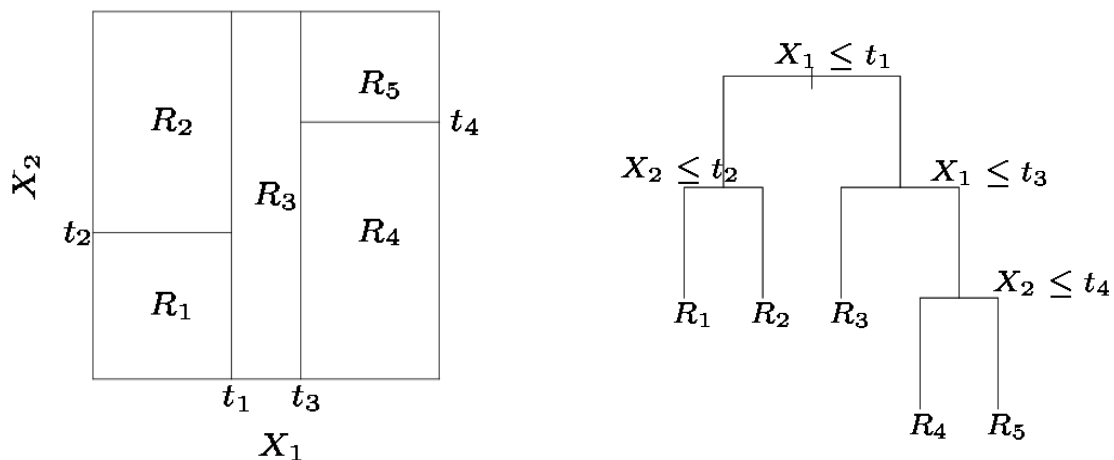


Figure 13: Decision Tree Example

The below example shows the complete flow of feature based component detection module until classification step. The word count program analyzes the important words/features based on the term frequency in the class words document. These features are then passed to Word2Vec [3] to build the feature-synonym vector which is used as training data for the classification algorithm. The model then predicts the feature label for each class using this training data.

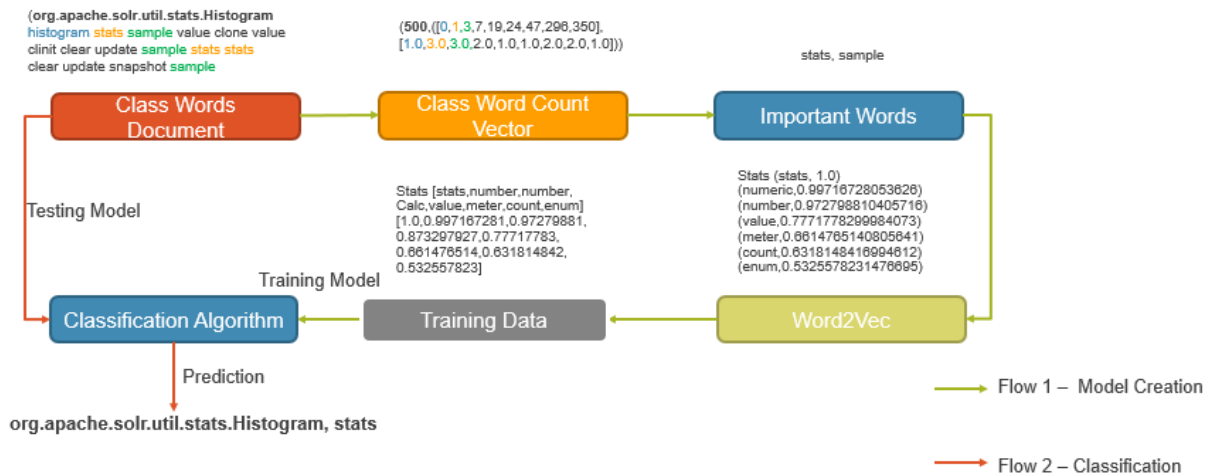


Figure 14: Classification Example

3.4.4 Component Detection

Supervised learning algorithm predicts the feature label for each class and outputs a tuple (class, feature). The classes with the same feature are then grouped together to form components. Since the components identified in the module are based on the features implemented in the classes, these components are called as feature based components. The output of this step is a tuple of the format (feature, <List of classes>). The below table gives an example of the components detected using this approach.

Table 5: Feature Based Component Example

Input tuple(class, feature)	Output tuple (feature , <List of classes>)
(org.apache.solr.ParseIntFieldUpdateProcessorFactory, processor)	(processor, org.apache.solr.ParseIntFieldUpdateProcessorFactory,
(org.apache.solr.ConcatFieldUpdateProcessorFactory, processor)	org.apache.solr.ConcatFieldUpdateProcessorFactory, org.apache.solr.StatelessUpdateProcessorFactory)
(org.apache.solr.StatelessUpdateProcessorFactory, processor)	(handler, org.apache.solr.StandardRequestHandler,
(org.apache.solr.StandardRequestHandler, handler)	org.apache.solr.BinaryUpdateRequestHandler)
(org.apache.solr.BinaryUpdateRequestHandler, handler)	

3.4.5 Connector Identification

Connector is a directed dependency relation between components. If any class A in component-1 is dependent on any class B in component-2, then a directed relationship is established between these components from component-1 to component-2. The degree of dependency is identified using the class dependency matrix generated in the data extraction step. This degree of dependency is then used to analyze the dependencies between the classes in different components. The connector could be unidirectional or bidirectional depending on the calling relationships among the classes in different components.



	BinaryRequest Handler	StandardRequestHandler	solrRequest	documentAnalysisRequest
BinaryRequestHandler	100.0	20.0	15.90	32.22
StandardRequestHandler	0.0	100.0	45.0	0.0

Figure 15: Connector Example

3.5 Dependency Based Component Detection

Dependency based component detection uses an unsupervised learning approach for identifying the components based on the calling relationships between the classes. The class dependency matrix generated in the data extraction module is used for building the class dependency vector. The class dependency vector gives the degree of dependency of a class on all the classes in the project. If a class A is not dependent on another class B, then the degree of dependency will be zero. The degree of dependency on itself will be given as 100.0.

Sample class dependency vector:

org.apache.solr.store.hdfs.HdfsDirectory,0.0,0.0,33.33333333333335,67.33333333333335,100.0

Once the vectors are generated, K-Means [3] clustering algorithm is used to cluster the classes into components. K-Means is an unsupervised learning algorithm used for clustering the data. K-Means algorithm classifies the data into a given number of clusters (k) by defining the centroids for each cluster. To ensure best clustering these centroids should be placed as far away as possible from each other. Once the centroids are determined, each point in the feature space is assigned to its nearest centroid. Once

the clusters are identified, we to recalculate the centroid for each cluster. Then the data points are remapped to their nearest centroids. This process is iterated until there is no change in the centroids.

The number of clusters K in the K-Means clustering is unknown. To determine the k value, the K-Means algorithm is executed for different cluster sizes and the sum of squared error (SSE) is computed. The K -value that gives the least sum of squared distance or the k -value at which there is not much difference in the SSE is chosen as the final cluster size.

The output of the K-Means will be tuple (class, component-number). The classes with the same component prediction are grouped together and a new tuple is generated in the format (component-number, <List of classes>).

3.6 Component Specific Feature Extraction

Each component consists of many classes that implement a number of features. This step is used to identify the key features implemented by each of the components. The input for this step could be the components detected from supervised or unsupervised learning approaches. Features in the components are detected by using semantic analysis. The high level flow of feature extraction from the components is shown in figure 15. It has four steps. In the first step, the data from the class documents of all the classes within a component are aggregated to form the component words document.

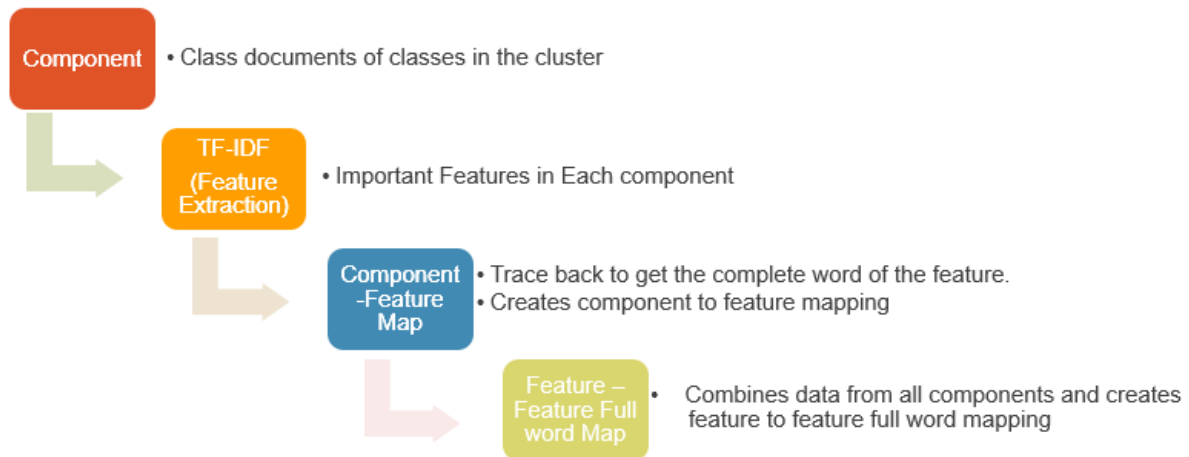


Figure 16: Component Specific Feature Extraction

The component words documents are then passed to the Term Frequency-Inverse Document Frequency (TF-IDF) for feature extraction. TF-IDF [3] is the product of term frequency and inverse document frequency. The term frequency is the number of times a word appears in a specific document. Document frequency is the frequency of the term in all the documents. The Inverse document frequency intends to reduce the importance of the word that occurs most frequently in all the documents. It is mainly used to eliminate the common terms across all the documents. The IDF value is computed by dividing the total number of documents with the number of documents that contain the given term t and then applying logarithm to the resultant value. If the term appears in more documents, it is more likely to be a common term that is not specific to any given document, hence the log value of the word reduces to zero ensuring that the IDF value and thereby the TF-IDF value is less for this term.

$$\text{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

- N : the total number of documents
- $|\{d \in D : t \in d\}|$: the number of documents in which the term t appears

The TF-IDF is calculated using the following formula:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \times \text{idf}(t, D)$$

TF-IDF value is high if the term has high term frequency and a low document frequency in the whole collection of documents. Hence by considering the TF-IDF value, we can eliminate the common terms in determining features.

For each component, the words are sorted based on the TF-IDF [3] values and top three features are extracted. The output of this step is a tuple of the format (Component, <List of Features>). Since the features are individual words obtained after splitting the class/method names based on camel casing, the words are traced back to obtain the full words from the class metadata documents. The full words are called as the feature variants. Once the feature variants are extracted, tuples of the format (feature, <List of feature variants>) are created. These tuples are then mapped to the component. Also while identifying the feature variants, the class from which the variant is obtained is determined to create a class to feature variant mapping.

It is possible that a feature is implemented in different components. The feature variants implemented by different components could be different. Hence we need to aggregate the (feature, <List of feature variants>) tuples generated from all the components to obtain the final feature to feature variant mapping. Following example shows the mapping created at each step of this module.

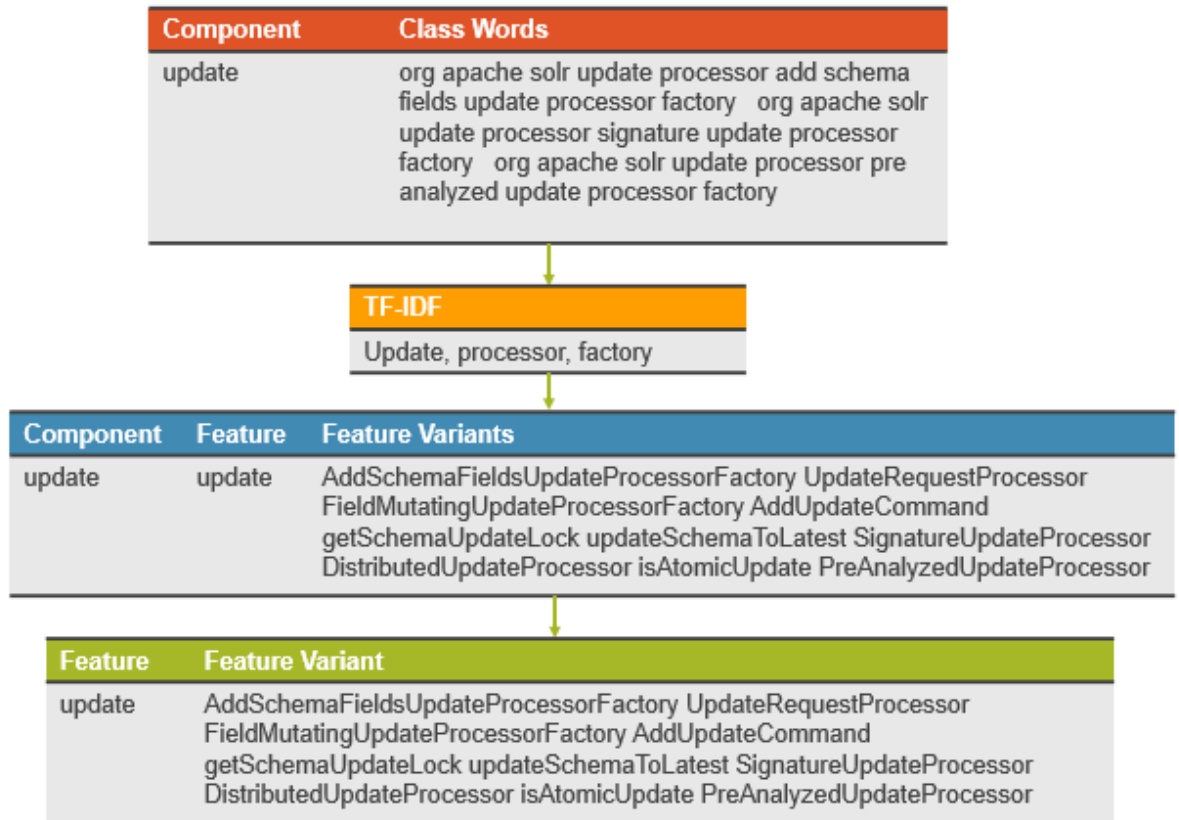


Figure 17: Feature Extraction Example

3.7 Ontology and Query Processing

Ontology is the description of concepts and the relationships that exists between them. It deals with identifying the entities, their relationships and representing them in a hierarchy such that similar entities are grouped together. Software Ontology [17] is a way of describing the software in terms of information like platform, version, types and properties of software components, features and relationships between them.

OWL [18] is a standard web ontology language proposed by W3C. It is an extension of RDF (the Resource Description Framework). RDF is an assertion model to represent the resources available on the web in the form of RDF triples of "subject", "predicate" and "object". However, RDF schema doesn't provide means to represent the relationships between classes and properties. OWL is used to represent

classes and their interrelationships. An OWL Class defines a group of individuals that share common properties. For example “Person” is a class. OWL Individual is an instance of the class. For example “John” is an individual of the class “Person”.

OWL provides a richer set of vocabulary for defining properties and imposing restrictions on them. There are two types of properties in OWL, Datatype properties and object properties. Object properties represent relationships among individuals and datatype properties are used to assign data values to individuals. For Example: “hasParent” is an Object property while “hasAge” is a data property. In OWL, we can define properties as reflexive, transitive, symmetric, functional, and inverse. OWL class properties like equivalent class, subclass of, disjoint class can be used to represent the relationships between classes. Thus various relationships and restrictions can be imposed on the classes in OWL. OWL is mainly used to provide meaningful descriptions to the terminology and the information on the web to enable reasoning on the web documents.

OWL API [23] is an open source java API for importing, creating and manipulating OWL ontologies. It is primarily maintained by University of Manchester. The latest version of OWL API is based on OWL 2 specification which includes OWL-Lite, OWL-DL and some elements of OWL-Full. They are all sub languages of OWL with some variations and differences in restrictions. OWL-Lite is more restrictive compared to OWL-DL and OWL-Full. It allows representation of hierarchies with some constraints. OWL-Full is syntactically similar to RDF and it doesn’t have any restrictions like OWL-Lite.

This thesis aims at generating a model that represents the software project which can be queried to search for functionality. We have used OWL 2.0 Ontology to represent the project. The ontology of a project is built using the information generated in the previous modules. Software ontology [22] is used as a base for generating the project ontology. Software ontology is developed at Information Sciences Institute in University of South California. It is a resource for describing the aspects of a software architecture like software types, operating system of the software, programming

languages, software components, packages, domains, etc. This ontology is most suitable for our requirement as it provides a means to describe most of the concepts and their relationships in software architecture. We have extended this ontology as per our requirements. The following figure 17 represents the software ontology as visualized in protégé using OWL Viz.

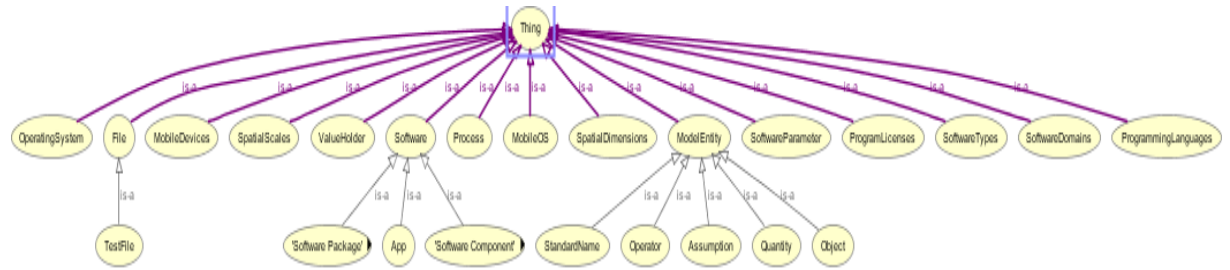


Figure 18: Software Ontology

FeatureDomain class is added as a subclass to software domains class. All the features discovered in the feature extraction process are added as subclasses to the feature domain class. Each feature has a set of feature variants. The feature variants are connected to their respective features using subclass relationship.

Software ontology has a software component class. All the components detected using supervised learning approach are added as subclasses to this class. Each component has a set of classes. All the classes within a component are connected to the component using a subclass object property.

The component to feature mapping obtained in the component specific feature extraction module is used to establish the relationship between the components and features. An object property “hasComponent” is created. And all the components implementing a specific feature are connected to this feature using this property. Similarly all feature variants are connected to their respective classes using the class to feature variant mapping. Figure 18 displays the steps involved in generating the ontology for a given model using the different outputs generated in the previous modules.

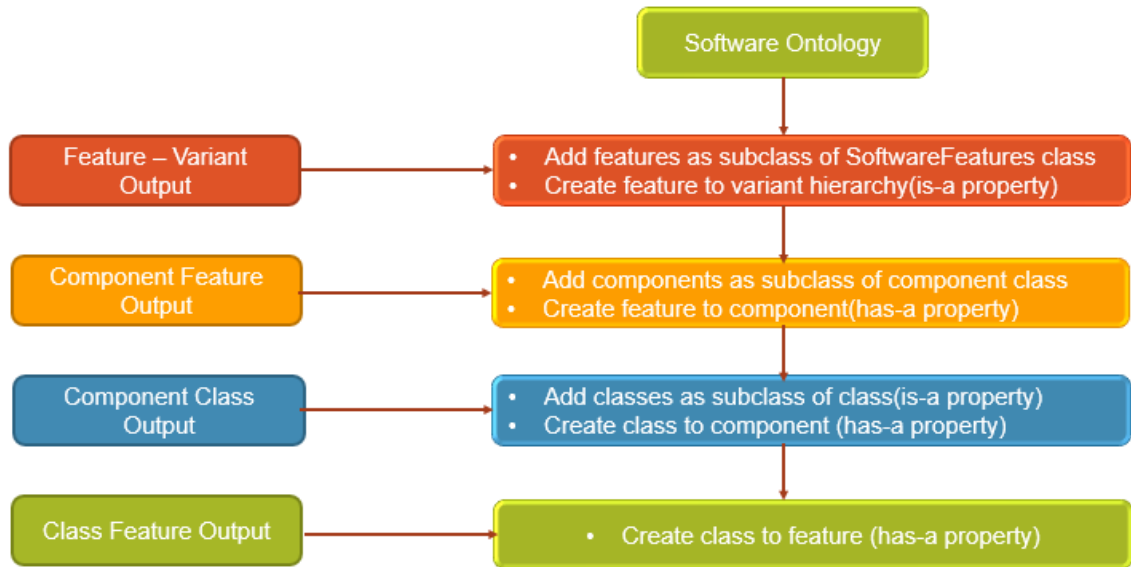


Figure 19: Ontology Generation

The generated ontology is visualized using Protégé [4] tool. Protégé is an open source ontology editor developed at Stanford University in collaboration with University of Manchester. It provides an interface to define ontologies, query ontologies and visualize them. OWL Viz [19] is used to represent the class hierarchy in Protégé. OWL Viz is a plugin that depends on Graphviz [21] that is a visualization software that represents structural information as diagrams of abstract graphs and networks. Ontograp [20] is a plugin used for visualizing the object properties and the relationships between various classes in the ontology. It provides a search functionality using which we can search for any feature and it displays all the features with that name and the components/classes implementing that feature.

One of the key advantages of building an ontology to represent the software is that the ontology can be queried using any RDF querying language like SPARQL [5]. SPARQL is a semantic query language used for retrieving and manipulating data stored in RDF format. Since OWL is an extension of RDF. SPARQL can be used to query OWL Ontology. There are two ways for querying OWL Ontologies:

- 1) Using Jena API and writing a java program with embedded SPARQL query/create an interface to provide the SPARQL query

- 2) Using SPARQL plugin in Protégé

In this thesis, we have used the SPARQL plugin in Protégé to query the generated ontology and do some reasoning on the same. Following are some of the use cases for query analysis:

- 1) Get the list of Features
- 2) Identify components implementing a feature
- 3) Get the list of variants for a feature
- 4) Identify the classes implementing a feature variant
- 5) Get the connections/dependencies between components

CHAPTER 4

RESULTS AND EVALUATION

This thesis builds a model of a software project from the source code dynamically. For building the model, it analyzes the features, components and connectors. Two types of components are detected using the proposed solution. Feature based components are groups of classes implementing similar features. Dependency based components are groups of classes that have a higher degree of dependency. Component definition changes based on perspective. So depending on the developer's use case, one could use the respective component structure for their analysis. The proposed model also identifies the connectors which are directed relationships between the components. Two projects are analyzed as part of case studies: Apache Solr and Apache Lucene. The following sections present the results from both the projects and the interaction between the components in these projects.

4.1 Apache Solr

Apache Solr is an open source search platform built using Apache Lucene and written in Java. Apache Solr is used as a case study for the approach proposed in the thesis. Apache Solr is a highly reliable and scalable open source search server. Below are the statistics of the Apache Solr project generated using Java call graph. Solr has 836 classes and lots of features and dependencies which makes it a good candidate for feature based and dependency based analysis using our proposed model. The key assumption for the feature based analysis is that the names used for code entities reflect their functionality and follow camel case.

Table 6: Apache Solr Statistics

Entity	Number
Dependencies	84692
Packages	176
Classes	836
Methods	8396
Parameters	1812

Apache Solr has 836 classes with many methods and dependencies. Java call graph project is used for identifying all the static dependencies in the given software project. Below is the sample output from the call graph.

Table 7: Sample Call graph output

ClassName:MethodName\$parameter	(typeOfCall)ClassName:MethodName\$parameter
M:org.apache.solr.handler.component.FieldFacetStats:getStatsValues	(l)java.util.Map:get
M:org.apache.solr.handler.component.FieldFacetStats:getStatsValues	(S)org.apache.solr.handler.component.StatsValuesFactory:createStatsValues
M:org.apache.solr.handler.component.FieldFacetStats:getStatsValues	(l)org.apache.solr.handler.component.StatsValues:setNextReader
M:org.apache.solr.handler.component.FieldFacetStats:facet	(M)org.apache.lucene.queries.function.FunctionValues:exists
M:org.apache.solr.handler.component.FieldFacetStats:facet	(M)org.apache.lucene.queries.function.FunctionValues:strVal
M:org.apache.solr.handler.component.FieldFacetStats:facet	(O)org.apache.solr.handler.component.FieldFacetStats:getStatsValues
M:org.apache.solr.handler.component.FieldFacetStats:facet	(l)org.apache.solr.handler.component.StatsValues:accumulate

Below table shows the call graph statistics for a sample set of classes in Apache Solr. It displays the number of methods, attributes, calling classes, calling methods, etc for a class.

Table 8: Sample Callgraph Statistics

Class	Methods	Attributes	Calling Classes	Calling Methods
org.apache.solr.core.SolrCore	95	15	85	96
org.apache.solr.search.SolrIndexSearcher	94	16	43	59
org.apache.solr.schema.IndexSchema	80	12	20	40
org.apache.solr.cloud.ZkController	71	12	40	104
org.apache.solr.search.ExtendedDismaxQParser	68	14	17	40
org.apache.solr.schema.FieldType	65	3	14	36
org.apache.solr.cloud.OverseerCollectionProcessor	63	14	43	131
org.apache.solr.cloud.Overseer	58	15	25	67
org.apache.solr.update.UpdateLog	55	14	26	56
org.apache.solr.handler.component.ResponseBuilder	54	2	12	23
org.apache.solr.handler.SnapPuller	53	21	31	50
org.apache.solr.core.CoreContainer	47	3	24	79
org.apache.solr.core.ConfigSolr	47	3	12	13
org.apache.solr.update.TransactionLog	46	8	11	21
org.apache.solr.util.ConcurrentLRUCache	46	8	0	0

4.1.1 Feature Based Analysis

The proposed model discovers two types of components: feature based components, dependency based components. Feature based components are groups of classes that implement similar features. Features in the given project are detected using semantic analysis. For semantic analysis, two sets of input data are generated. The class only documents consist of all the words related to the class like the package name, class name, method names, and attribute names. Class+Dependency data consist of all words related to class like package name, class name, method names, attribute names, calling class names, calling method names. The following figure shows the difference in the content of the class only and Class+Dependency input data.

Class Only	Class + Dependency
Package Name	Package Name
Class Name	Class Name
Method Names	Method Names
Method Parameters	Method Parameters
Attribute Names	Attribute Names
	Calling Class/Interface Name
	Calling Method name
	Calling method parameters

Figure 20: Class Only vs Class + Dependency Input Data

Both the class documents are analyzed to identify the features and components using the same set of steps. Firstly, each of these words in the class documents are split using camel case naming convention and then lemmatized to convert the words to their base form. Without lemmatization the words “values” and “value” are treated as different features. Since the approach uses semantic analysis, it is required to convert all the words to their base forms for more accuracy. Also, stop words like “the”, “get”, “set” are removed during lemmatization of the class documents. Hence for identifying the important features, we could use word count. Word count is basically the frequency of terms in the

class document. From each class document, the top two words are taken based on the decreasing order of frequency. The following figure shows the set of important words/features identified for Apache Solr project using the Class+Dependency documents.

Word	Count
schema	4225
param	4096
init	3908
request	3551
field	3405
update	3226
search	2902
cloud	2774
name	2740
response	2511
query	2306
list	2057
handler	1892
exception	1888

Word	Count
add	1705
index	1487
processor	1146
component	1128
value	964
map	948
parser	944
factory	924
client	917
reader	458
plugin	451
loader	448
spelling	425

Word	Count
directory	412
stream	405
slice	316
rest	297
controller	297
distribute	293
delete	282
clinit	276
function	275
collection	391
sort	384
facet	273

Figure 21: Apache Solr Important Words

During development, different words may be used to represent the same functionality. For example output and result represent the same function. Hence it is required to identify the synonyms of the important words identified using word count program so that classes that contain the important word or their synonyms can be grouped together for identifying the feature based components. As mentioned before, Word2Vec [3] is a machine learning algorithm that represents words in the form of distributed vectors and identifies the similarity between the words using cosine distance. Thus synonyms can be detected based on the cosine similarity between the words. The important words and their synonyms are used to create the feature synonym data. This data is used for training the supervised learning algorithms. The class documents are then used for testing the algorithm which classifies the classes. The classification of the classes is tested using two supervised learning algorithms: Naïve Bayes and Random Forest. It is observed that the accuracy of Naïve Bayes (87.16%) is more

compared to the Random Forest (68.75%) algorithm for semantic analysis. Below tables show the classification output using Naïve Bayes algorithm for class only and Class+Dependency documents.

Table 9: Naïve Bayes Classification Output for Class only Data

Class	Feature
org.apache.solr.rest.SolrSchemaRestApi	schema
org.apache.solr.util.plugin.NamedListPluginLoader	plugin
org.apache.solr.cloud.LeaderInitiatedRecoveryThread	leader
org.apache.solr.search.Grouping	facet
org.apache.solr.spelling.AbstractLuceneSpellChecker	diagnostic
org.apache.solr.handler.component.DateStatsValues	stats
org.apache.solr.update.processor.RemoveBlankFieldUpdateProcessorFactory	processor
org.apache.solr.util.RTimer	diagnostic
org.apache.solr.update.processor.StatelessScriptUpdateProcessorFactory	processor
org.apache.solr.common.cloud.HashBasedRouter	hash
org.apache.solr.update.processor.CloneFieldUpdateProcessorFactory	processor
org.apache.solr.common.cloud.RoutingRule	route
org.apache.solr.search.SpatialFilterQParserPlugin	filter
org.apache.solr.schema.BBoxField	field

Table 10: Naïve Bayes Classification output for Class + Dependency Data

Class	Feature
org.apache.solr.rest.SolrSchemaRestApi	exception
org.apache.solr.util.plugin.NamedListPluginLoader	util
org.apache.solr.cloud.LeaderInitiatedRecoveryThread	permit
org.apache.solr.search.Grouping	collector
org.apache.solr.spelling.AbstractLuceneSpellChecker	checker
org.apache.solr.handler.component.DateStatsValues	list
org.apache.solr.update.processor.RemoveBlankFieldUpdateProcessorFactory	processor
org.apache.solr.util.RTimer	util
org.apache.solr.update.processor.StatelessScriptUpdateProcessorFactory	processor
org.apache.solr.common.cloud.HashBasedRouter	node
org.apache.solr.update.processor.CloneFieldUpdateProcessorFactory	processor
org.apache.solr.common.cloud.RoutingRule	cloud
org.apache.solr.search.SpatialFilterQParserPlugin	plugin
org.apache.solr.schema.BBoxField	closable
org.apache.solr.request.SolrRequestHandler	request
org.apache.solr.core.CloseHook	post
org.apache.solr.update.processor.DocExpirationUpdateProcessorFactory	collector

The classification algorithm classifies the classes based on the features. The output of the classification algorithm is a tuple (class, feature). Using this output, classes are grouped together based on the feature identified during classification. Since these components are determined based on features, these set of components are called feature based components. Naïve Bayes classification generated 136 components using Class+Dependency data and 167 components using class only data. Below table displays a sample of feature based components detected using class +Dependency data.

Table 11: Feature Based Components using Class+Dependency data

Component	Classes
param	org.apache.solr.common.params.SolrParams org.apache.solr.common.params.ModifiableSolrParams org.apache.solr.common.params.CoreAdminParams org.apache.solr.common.params.MultiMapSolrParams org.apache.solr.common.params.MapSolrParams org.apache.solr.highlight.SimpleBoundaryScanner org.apache.solr.common.params.RequiredSolrParams
similarity	org.apache.solr.search.similarities.LMJelinekMercerSimilarityFactory org.apache.solr.rest.schema.analysis.FSTSynonymFilterFactory org.apache.solr.search.similarities.LMDirichletSimilarityFactory org.apache.solr.search.similarities.DefaultSimilarityFactory
Writer	org.apache.solr.response.BinaryQueryResponseWriter org.apache.solr.response.PythonWriter org.apache.solr.response.NaNFloatWriter org.apache.solr.response.PHPSerializedWriter org.apache.solr.response.PHPResponseWriter org.apache.solr.response.JSONResponseWriter org.apache.solr.response.XMLResponseWriter org.apache.solr.response.SchemaXmlResponseWriter org.apache.solr.response.RubyWriter org.apache.solr.response.PHPSerializedResponseWriter org.apache.solr.response.PHPWriter org.apache.solr.response.QueryResponseWriter org.apache.solr.internal.csv.writer.CSVWriter org.apache.solr.response.PythonResponseWriter org.apache.solr.response.RubyResponseWriter org.apache.solr.response.CSVResponseWriter
container	org.apache.solr.core.ZkContainer

Table 12: Feature Based Components using Class only data

Component	Classes
Param	org.apache.solr.common.params.SolrParams org.apache.solr.common.params.ModifiableSolrParams org.apache.solr.common.params.DisMaxParams org.apache.solr.request.SolrQueryRequest org.apache.solr.common.params.MoreLikeThisParams org.apache.solr.common.params.CommonParams org.apache.solr.common.params.MultiMapSolrParams org.apache.solr.common.params.MapSolrParams org.apache.solr.common.params.CollectionParams org.apache.solr.search.QueryParsing org.apache.solr.update.processor.FieldMutatingUpdateProcessorFactory org.apache.solr.common.params.SpellingParams org.apache.solr.common.params.RequiredSolrParams org.apache.solr.common.params.FacetParams
similarity	org.apache.solr.search.similarities.IBSimilarityFactory org.apache.solr.search.similarities.LMJelinekMercerSimilarityFactory org.apache.solr.schema.FieldTypePluginLoader org.apache.solr.search.similarities.SweetSpotSimilarityFactory org.apache.solr.search.similarities.BM25SimilarityFactory org.apache.solr.search.similarities.LMDirichletSimilarityFactory org.apache.solr.search.similarities.SchemaSimilarityFactory org.apache.solr.handler.loader.ContentStreamLoader org.apache.solr.handler.loader.JavabinLoader org.apache.solr.handler.loader.SingleThreadedCSVLoader org.apache.solr.handler.loader.CSVLoader
Writer	org.apache.solr.update.DefaultSolrCoreState org.apache.solr.internal.csv.writer.CSVConfigGuesser

Components identified using both the inputs are then used to detect the features. Each component implements a set of features. To identify the features, we aggregate the words from the class documents of all the classes in the component and then perform TF-IDF (Term Frequency – Inverse Document Frequency). The top three features of each component are taken based on the TF-IDF value of the words related to the component. These words form the component specific features. The words are then traced back to the original compound word from which it is extracted. The original word is

called as the feature variant. In this step, component to feature and class to feature variant mapping is generated. Component to feature map is the relation between the components and its set of features. Class to feature variant is the mapping between the class and the original compound word from which the feature is extracted.

Table 13: Sample component to feature mapping

Component	Feature	Feature Variants
Filter	parser	SpatialFilterQParserPlugin createParser QParserPlugin BlockJoinParentQParserPlugin createBJQParser ComplexPhraseQParserPlugin BlockJoinChildQParserPlugin ExtendedDismaxQParserPlugin DisMaxQParserPlugin MaxScoreQParserPlugin SpatialBoxQParserPlugin SurroundQParserPlugin
similarity	factory	IBSimilarityFactory LMJelinekMercerSimilarityFactory SweetSpotSimilarityFactory BM25SimilarityFactory LMDirichletSimilarityFactory SchemaSimilarityFactory

Table 14: Sample Class to feature variant mapping

Class	Feature	Feature Variants
org.apache.solr.response.SortingResponseWriter	field	getFieldWriters
org.apache.solr.response.SortingResponseWriter	field	DoubleFieldWriter
org.apache.solr.response.SortingResponseWriter	field	FieldWriter
org.apache.solr.response.SortingResponseWriter	field	FloatFieldWriter
org.apache.solr.response.SortingResponseWriter	field	IntFieldWriter
org.apache.solr.response.SortingResponseWriter	field	LongFieldWriter
org.apache.solr.response.SortingResponseWriter	field	MultiFieldWriter
org.apache.solr.response.SortingResponseWriter	field	StringFieldWriter
org.apache.solr.request.SimpleFacets	field	getFieldMissingCount
org.apache.solr.request.SimpleFacets	field	getFacetFieldCounts
org.apache.solr.request.SimpleFacets	field	getFieldCacheCounts
org.apache.solr.handler.component.FacetComponent	field	modifyRequestForFieldFacets

Each feature can have different types of variants. For example writer is a feature. Ruby Writer, Python Writer, JSON Writer, etc. are variants of the feature Writer. Feature to variant, component to feature and class to feature mappings are used for building a hierarchy of relationships. The proposed model also identifies the connectors between the components. Connectors represent the dependency relationships between the components. Below table shows a sample of connectors identified in Solr project.

Table 15: Sample Connectors in Solr

Component	Related Components
factory	request processor
similarity	property param parser plugin
transform	handler facet parser param

4.1.2 Dependency Based Analysis

Call graph provides the calling relationships between the classes. These relationships are analyzed to identify the dependency based components. This is based on the assumption that classes that are closely related to each other belong to the same component. Dependency based analysis produces components that contains classes which are highly cohesive. Degree of dependency is computed based on the frequency of interaction between the classes. For each class, a vector is created with the degree of dependency on all the classes. These vectors are then passed to K-Means [3] clustering algorithm. Below is the sample vector data passed to the clustering algorithm:

org.apache.solr.util.stats.Histogram	33.33333	33.33333	33.33333	0	0	0	0	0
org.apache.solr.spelling.SpellCheckCorrection	0	0	0	0	0	0	0	0
org.apache.solr.store.hdfs.HdfsDirectory	0	0	0	3.333333	3.333333	6.666667	6.666667	3.333333
org.apache.solr.highlight.DefaultEncoder	0	0	0	0	0	0	0	0
org.apache.solr.search.SolrIndexSearcher	0	0	0	12.09	7.45	0	0	0
org.apache.solr.handler.admin.SolrInfoMBeanHandler	0	0	54.79452	6.849315	0	0	0	4.109589
org.apache.solr.highlight.SolrFragmenter	0	0	0	0	0	0	0	0
org.apache.solr.internal.csv.CSVUtils	0	0	2.348993	1.006711	1.677852	1.006711	0	0
org.apache.solr.core.SolrCores	0	0	0	0	0	0	0	5.263158
org.apache.solr.rest.schema.BaseManagedTokenFactory	0	0	0	0	0	0	0	0
org.apache.solr.client.embedded.EmbeddedSolrServer	0	0	3.125	0	0	1.5625	0	4.6875
org.apache.solr.core.CoreContainer	0	0	0	0	0	0	0	0

Figure 22: Sample Data for Clustering

Before using K-Means [3] clustering technique, first we need to identify the value of k. K is the parameter that holds the number of clusters. To determine k-value, sum of squared errors (SSE) technique is used. SSE is a measure of sum of squared distance of all data points from their respective cluster centroid. The number of clusters is increased until the difference between the SSE values is less than a marginal value. A plot of the SSE values for different cluster sizes is shown in the below figure. The cluster size that results in minimum SSE value or the cluster size at which there is not much difference in the SSE value is considered as the best K-value. For Apache Solr project, based on the data from degree of dependencies, the optimal K-value is obtained as 108.

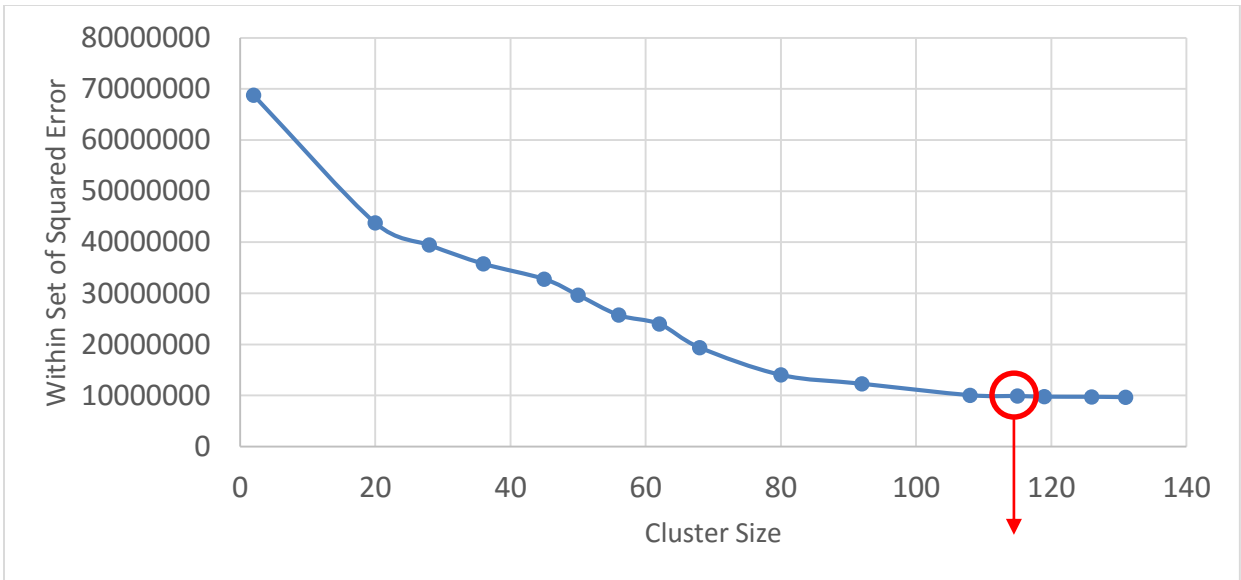


Figure 23: SSE Plot for different cluster sizes

Once the K-value is determined, the k value and the dependency data are passed as input to the train method in the K-Means [3] clustering algorithm in Spark MLlib. The algorithm then assigns a cluster number to each class in the input data. The classes are then grouped based on their cluster number to form components. Below is a sample of dependency based components detected using K-Means [3] clustering algorithm.

Table 16: Sample Dependency Based Components

Cluster No.	Classes
34	org.apache.solr.internal.csv.CSVUtils
96	org.apache.solr.common.util.JavaBinCodec
52	org.apache.solr.core.PluginInfo
4	org.apache.solr.schema.IntValueFieldType org.apache.solr.schema.FloatValueFieldType org.apache.solr.schema.DoubleValueFieldType
16	org.apache.solr.rest.schema.SchemaVersionResource
82	org.apache.solr.handler.StandardRequestHandler org.apache.solr.handler.RealTimeGetHandler
66	org.apache.solr.search.SpatialBoxQParserPlugin
28	org.apache.solr.common.params.CoreAdminParams
80	org.apache.solr.search.NamedParser

4.1.3 Evaluation

Manual analysis of Apache Solr project is conducted by Dr. Yongjie Zheng, Professor at University of Missouri Kansas city and his team. The manual architecture is created in the form of an xml document. The document has the hierarchy of components, features, feature variants and their relationships. We have used Java SAX parser to parse the xml and extract the information about components and their related features and feature variants. This information is used to create component document. Each of the words in the component document are split based on camel case naming convention and then lemmatized to convert the words to their base form. The transformed component document is then used to train the Naïve Bayes classification algorithm and generate a model. The model is then tested using the class documents generated from the Solr source code to predict the component label of each class. The classes are then grouped based on the classification label. Using the model generated using the data from manual architecture recovery, 115 components are identified. The components obtained from the manual analysis input are used for evaluating the components detected using feature based and dependency based analysis.

Table 17: Sample Components using Manual Architecture Recovery Input

Component	Classes
XSLTWriter	org.apache.solr.response.XSLTResponseWriter
NoOpDistributingUpdateProcessor	org.apache.solr.search.NoOpRegenerator org.apache.solr.update.processor.DistributingUpdateProcessorFactory org.apache.solr.update.processor.NoOpDistributingUpdateProcessorFactory
DistributedUpdateProcessor	org.apache.solr.update.processor.DistributedUpdateProcessorFactory
WritetoCache	org.apache.solr.search.CacheRegenerator org.apache.solr.store.blockcache.BlockDirectory org.apache.solr.store.blockcache.Cache org.apache.solr.search.CacheConfig
PrefixQueryParser	org.apache.solr.search.PrefixQParserPlugin org.apache.solr.schema.SpatialRecursivePrefixTreeFieldType org.apache.solr.schema.SpatialTermQueryPrefixTreeFieldType

For evaluating the components identified using different approaches, conditional probability function is used. It calculates the percentage of match between the components from different outputs.

$$P\left(\frac{CompA}{CompB}\right) = \frac{P(CompA \cap CompB)}{P(CompA)} * 100$$

$P(CompA \cap CompB)$ = Numbers of common classes between the two components

Once the percentage match is computed for all the components, the maximum percentage match for each component is determined. Below pie-charts show the comparison of component results using feature based analysis and dependency based analysis with the components identified using manual analysis data.

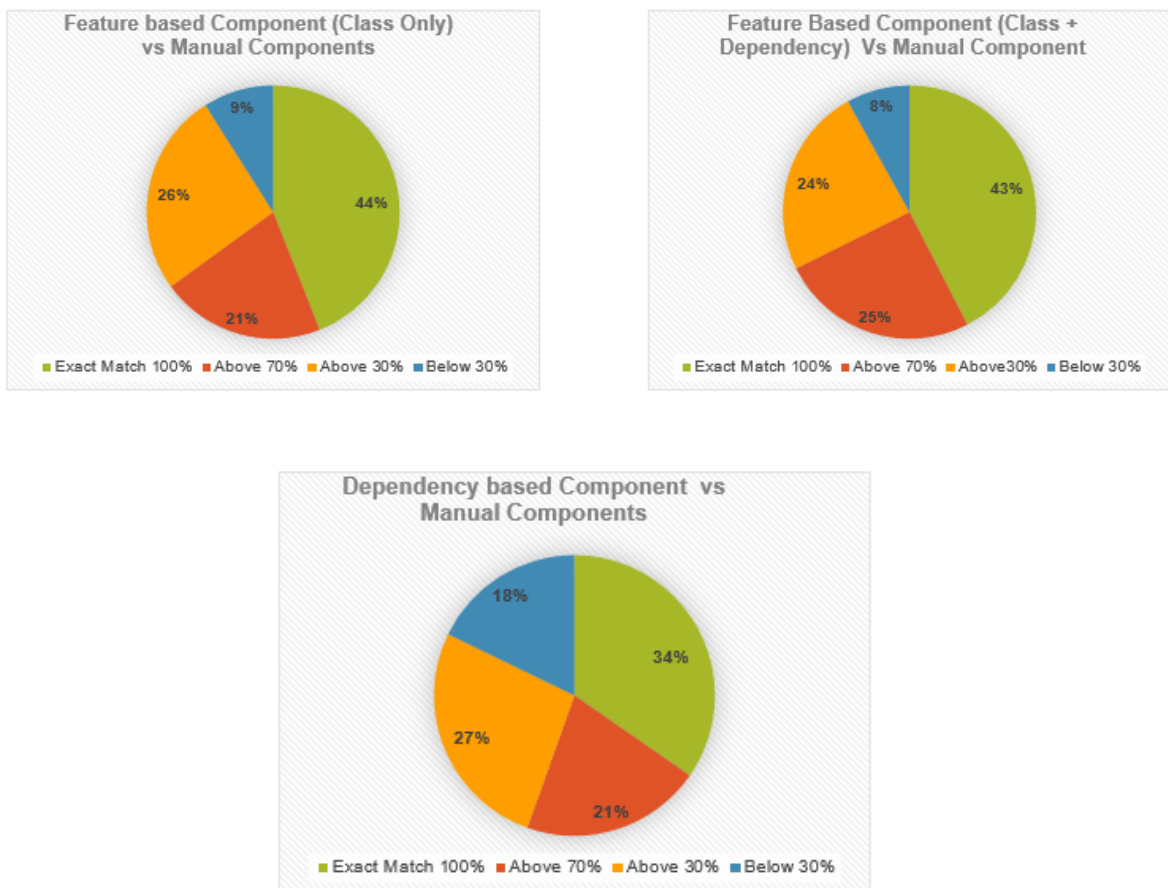


Figure 24: Component Evaluation with Manual Analysis

In Java, packages are generally created to group similar classes or related classes together. Hence packages themselves could be treated as components. Assuming, the developers have created packages appropriately to represent related classes, we could use the clusters of classes based on packages for evaluating the components generated using machine learning techniques. Below graphs show the comparison of components from feature based and dependency based analysis with the package based clusters.

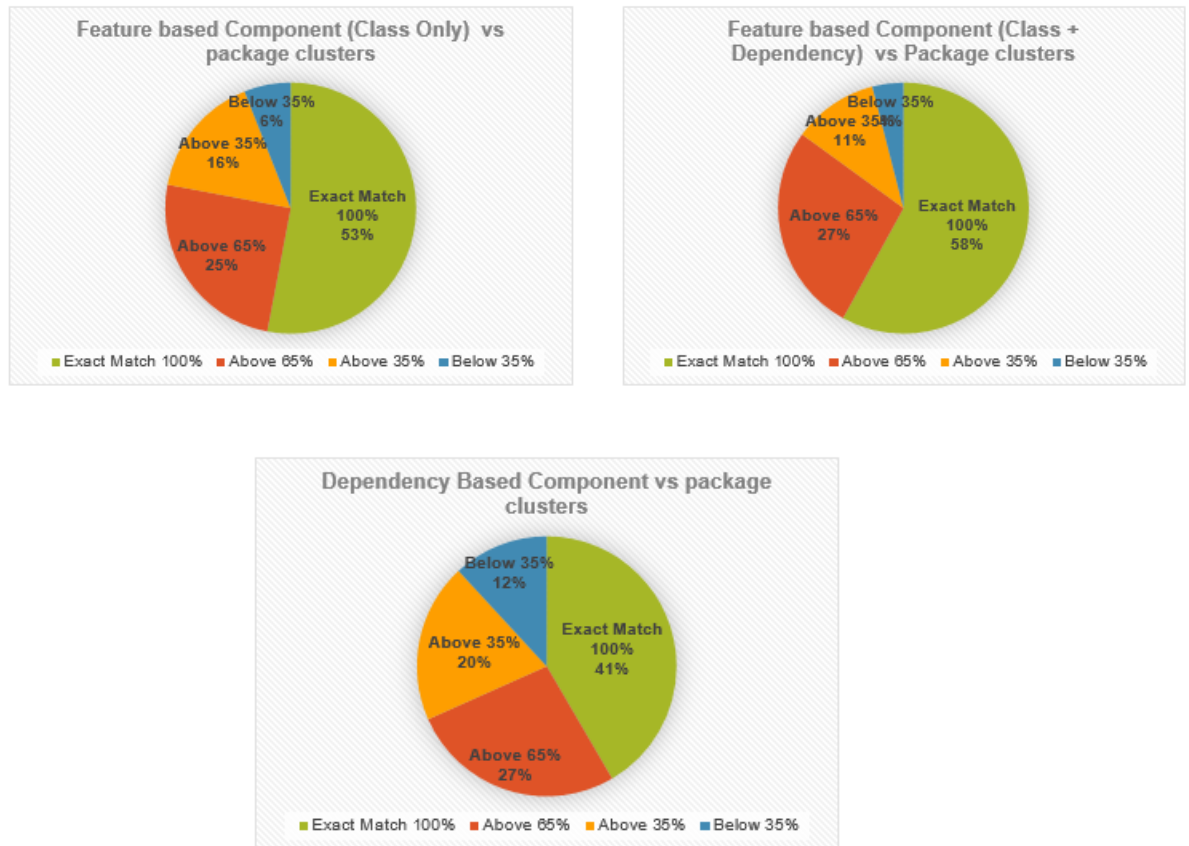


Figure 25: Component Evaluation with Package Cluster

From the above pie-charts it can be observed that the components obtained from semantic analysis are more similar to the components obtained using manual architecture recovery as well as package based clusters. The components obtained using Class + Dependency data have a better match with package clusters compared to the components obtained using class only data.

Solr has many features. In manual architecture recovery process, 28 features and 143 feature variants are detected. The features detected in the manual analysis are used for evaluating the features identified using the machine learning approach. But the features cannot be compared directly as the features recovered in manual analysis are given compound names e.g., Variant - Response Writers, Variant - Update Handlers etc. In machine learning approach, the features identified are single words. For example handler, writer, parser etc. So for evaluation we cannot make a perfect match between the manual features and automated features. Hence the features are evaluated using partial match approach. It is observed that out of 28 features from manual analysis 22 are identified using class only data, 24 are identified using class + dependency data and 26 are identified using dependency based analysis. Below graph represents the comparison of features identified from different approaches with the features from manual analysis. Nearly 90% of the features are recovered using the dependency based analysis of the proposed model. Also many new features are discovered using the automated approach.

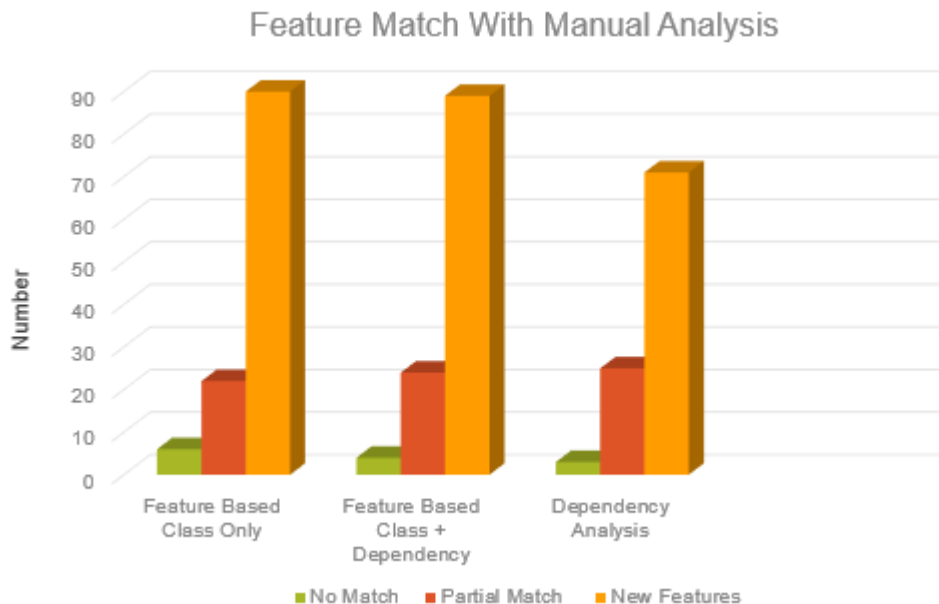


Figure 26: Feature evaluation of different approaches with Manual Analysis

The below table shows the partial match between features identified from machine learning approach and manual analysis.

Table 18: Partial Match Results of Features

Feature from Automated process	Matching feature from Manual Analysis
Parser	Variant - Query Parsers
Loader	Variant - Update Loaders
Facet	Query - Faceting
Highlight	Query - Highlighting
Suggest	Query - Suggest
Handler	Variant - Query Handlers
Spell	Query - Spell Checking
function	query - function queries
analysis	Query - analysis
Writer	Variant - response writers
command	variant - update commands
index	Variant - run index query commands
client	Variant - service clients
processor	Variant - update processors
post	update - post tool
update	Variant - update handlers
component	Variant - query components
search	Query - spatial/geospatial search
param	Variant - solr parameters
word	update - extract pdf/word
boost	Query - query boosting/elevation
analysis	query - terms reporting and analysis
stat	query - statistics collection
cache	Caching

Following are the features from manual analysis that are not discovered in automated approach:

- 1) query - clustering
- 2) query - morelikethis
- 3) query - nosql features

Following table shows the list of new features identified using the machine learning approach.

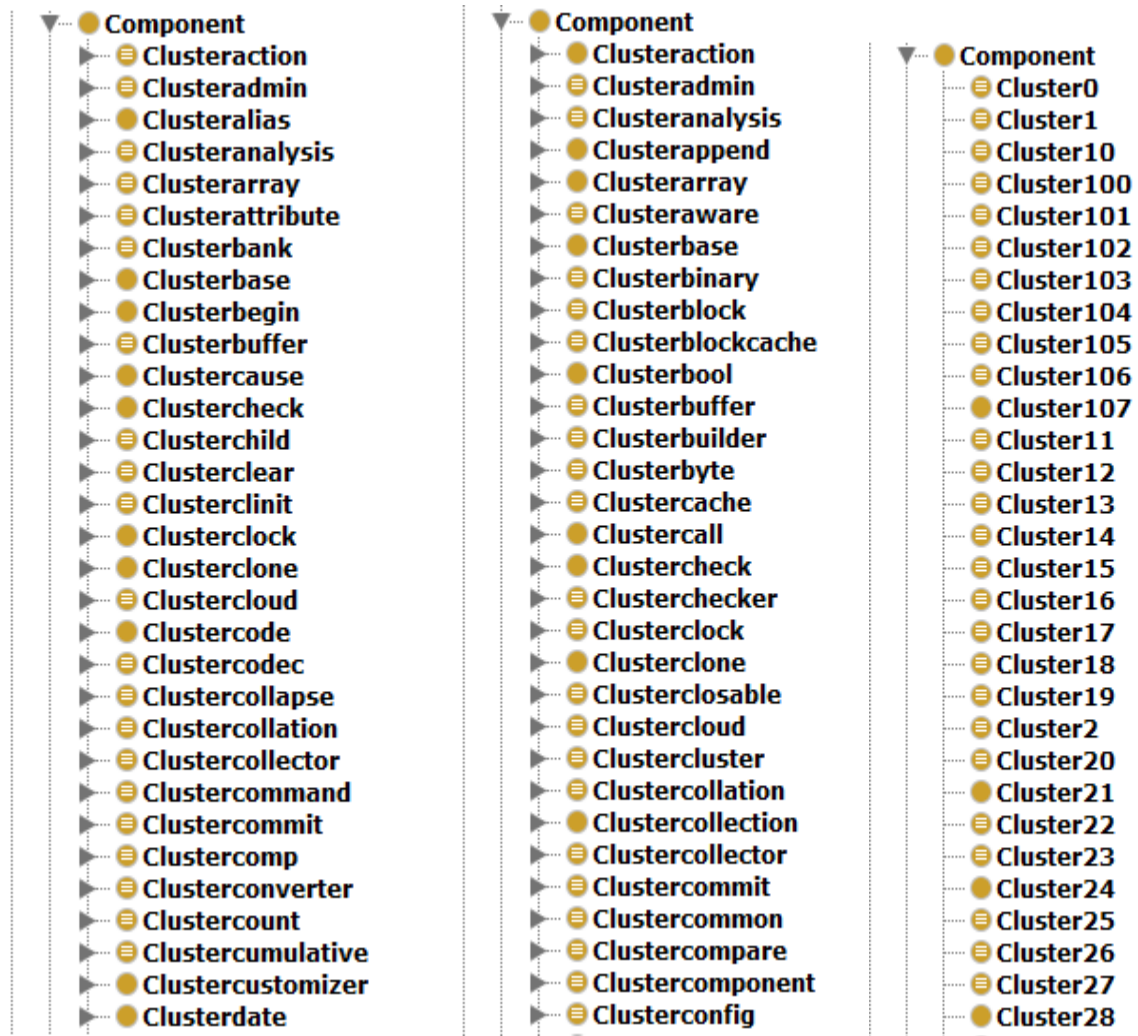
New Features	New Features	New Features
watcher	locater	sort
router	field	stream
response	thread	redirect
filter	factory	monitor
replicate	config	collection
transforme	server	queue
schema	executor	recovery
store	blockcache	event
fragmenter	csv	scanner
log	distance	resolver
buffer	servlet	formatter
cloud	standard	plugin
hdfs	token	similarity
dispatch	overseer	exception
xml	distributed	grouping

Figure 27: New Features discovered using proposed model

4.1.4 Ontology and Query Processing

One of the key goals of this thesis is to build a model that represents the software system accurately. For building the model OWL ontology is used. OWL provides various properties like sub class, equivalent class etc. to represent the relationships between different entities. Using the features, variants, components, classes, connectors detected in the automated approach, an ontology is generated. The generated ontology provides is-a relationship between features and variants, has-a relationship between features and components to identify the components that implement a feature, has-a relationship between variants and classes to identify the class that implements a particular feature variant, has-a relation among related components to represent the dependency relationship between the components. The generated ontology can be visualized using Protégé tool. OntoGraf is a plugin used to visualize the different entities of the software system and their relationships. OntoGraf also provides a search functionality through which we can search for any feature and the feature and its related components are displayed.

The following figure shows the hierarchy of components generated using different approaches: feature based analysis using class only data, feature based analysis using class+ dependency data, dependency based analysis. In feature based analysis, a classification algorithm is used to identify the key feature, hence the component name is same as the feature label. In dependency based analysis, cluster number is assigned for classes using K-Means clustering. Hence the component names in dependency based analysis have cluster numbers.



1. Feature based Class only 2. Feature based class + Dependency 3. Dependency based components

Figure 28: Components generated using different approaches

Components are basically groups of similar classes. Hence each component in the ontology has a set of classes. The components can be expanded in OntoGraf to visualize the classes present in it as shown in the figure below:

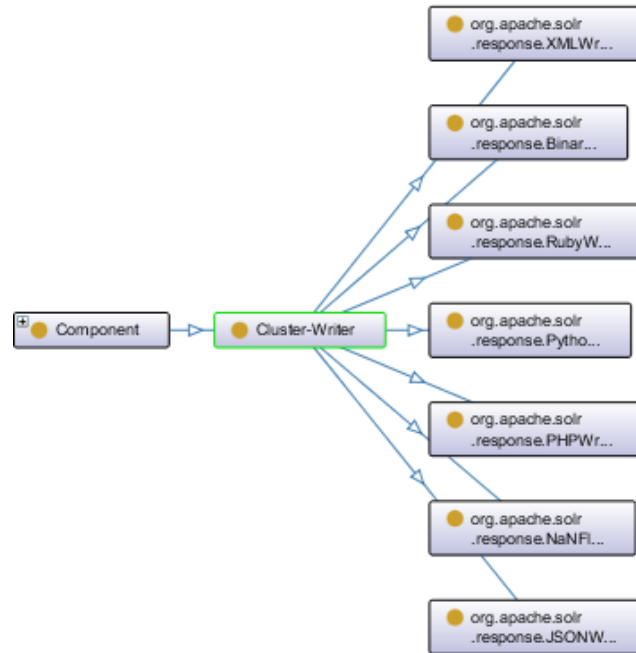


Figure 29: Component to class visualization

Apache Solr has many features. 28 key features are discovered in the manual architecture recovery process. In the automated approach 90 features are discovered using feature based analysis of class + dependency data, 89 features are discovered using feature based analysis of class only data and 71 features are discovered using dependency based analysis approach. All the features discovered in the automated approach are included in the generated ontology. The following figure shows the visualization of features discovered from manual analysis and dependency based automated approach.

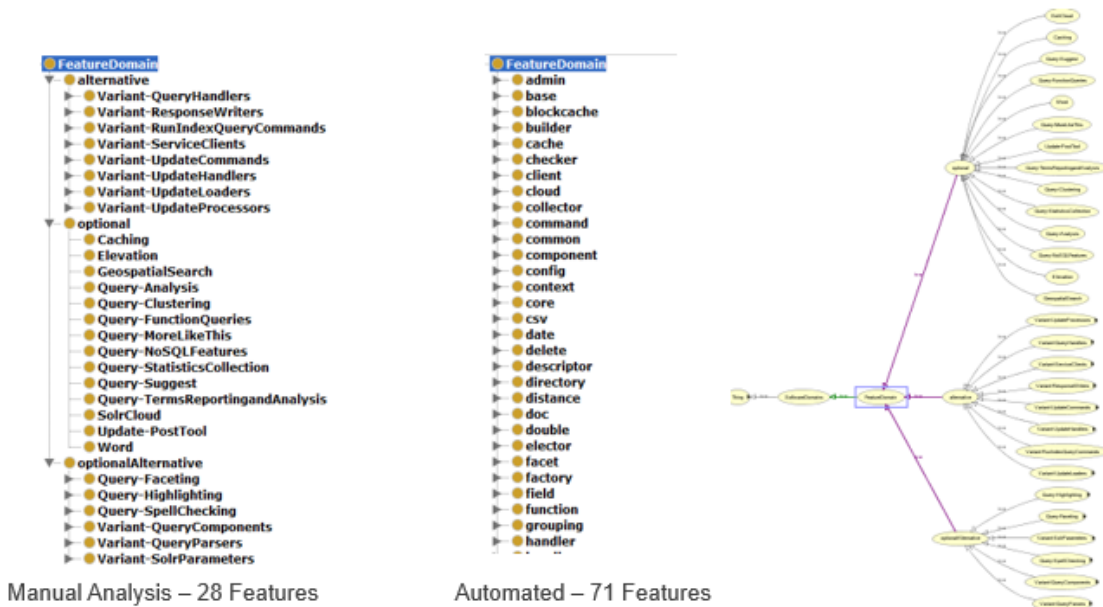


Figure 30: Ontology - Features

We can select any feature and expand the feature to visualize all the variants of the feature and the components that implement the feature. The following figure shows a sample feature Writer and its variants and components that implement that feature.

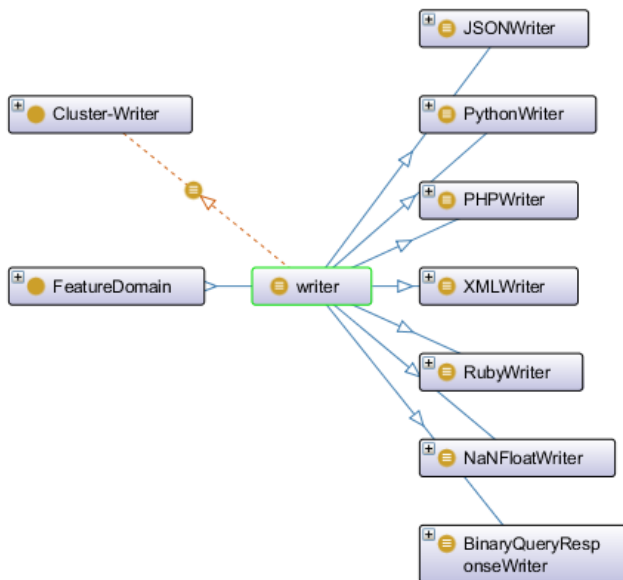


Figure 31: Feature - Feature Variant - Component Map

The feature to component graph could be further expanded to identify the classes that implement each variant of a given feature. The following figure shows a sample feature writer, sample component cluster-writer and their respective feature variants, classes and the relationships between them.

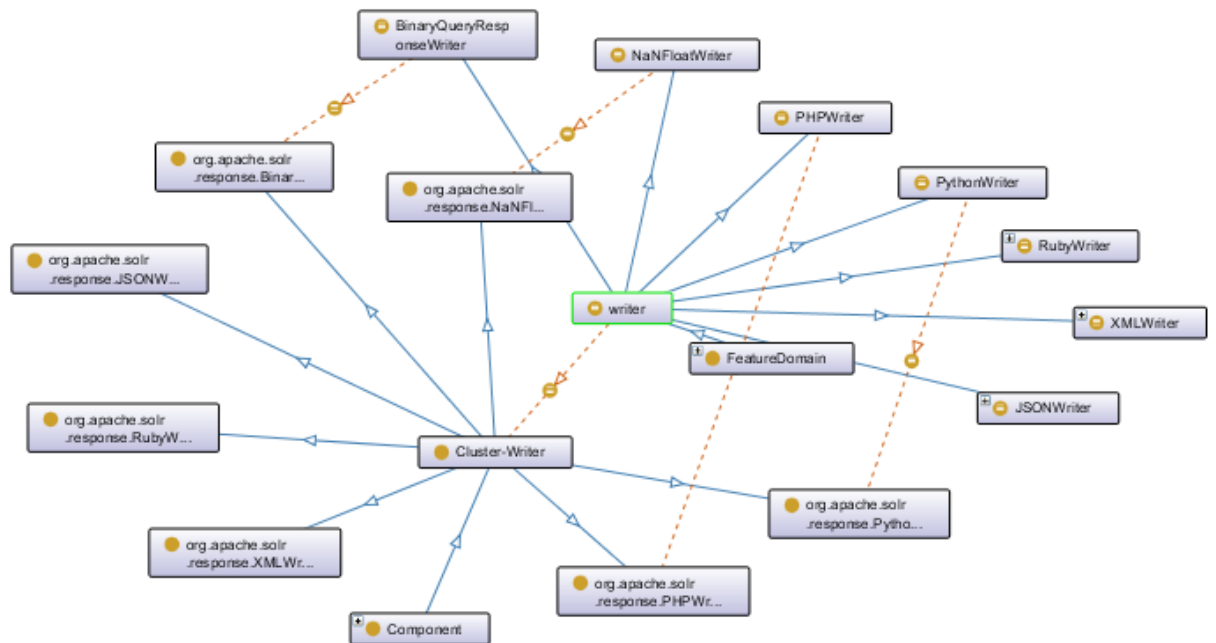
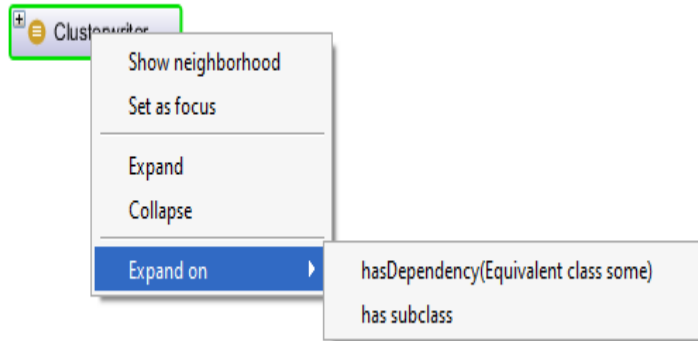


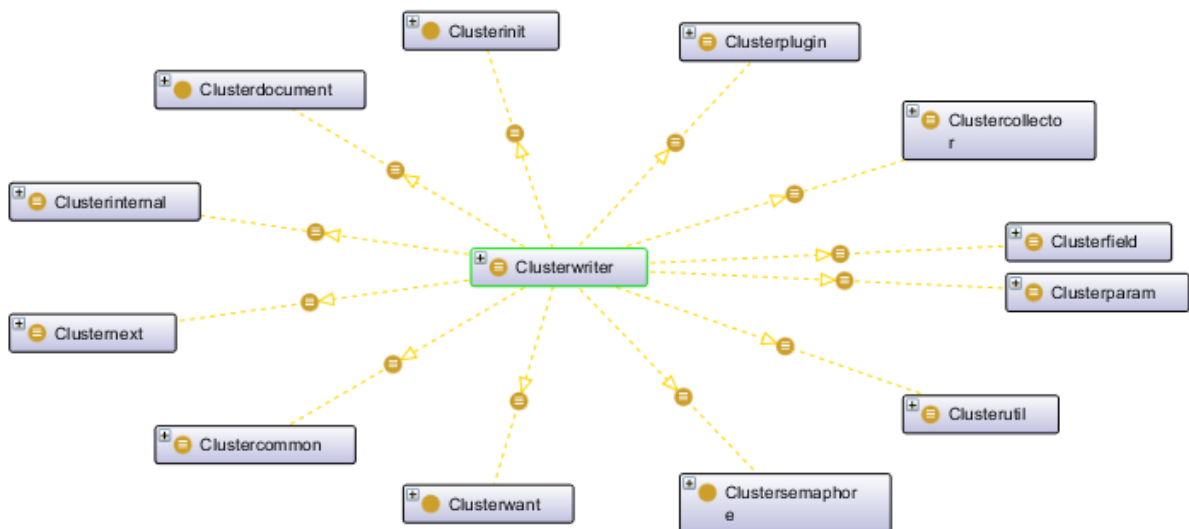
Figure 32: Class to feature variant visualization

The proposed model also identifies the connectors that are the dependency relationships among the components based on the callgraph data. The connectors are presented in ontology by creating hasDependency object property. It is an equivalent class property. If any of the classes in a component A are dependent on any of the classes in component B, then the two components are connected using the hasDependency property. It is a directed relationship from the calling component to the callee component. In OntoGraf any component can be expanded to identify its dependencies.

For visualizing the dependencies in OntoGraf, we can select the component from the component hierarchy and then right click on the component. It displays a “expand on” option that shows all the properties of this particular component. To get the list of dependencies, select the hasDependency equivalent class property.



The below figure shows a sample component and its dependencies using hasDependency property. The property is represented using a dotted line with the direction towards the callee component.



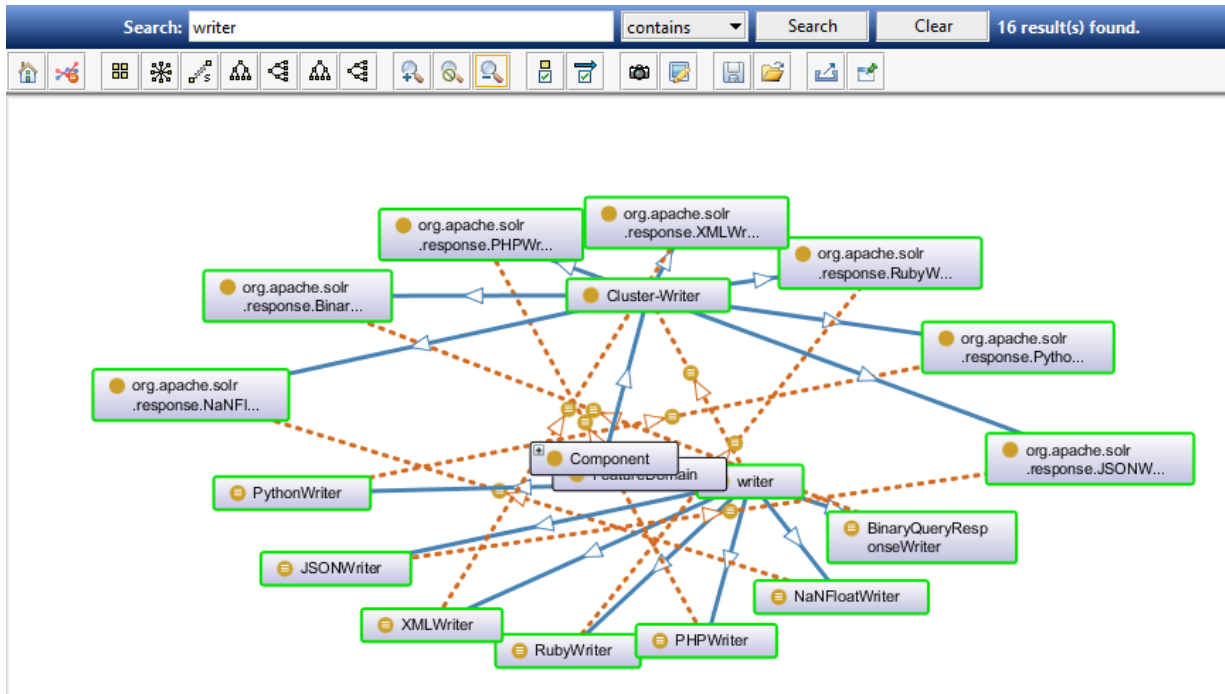


Figure 35: OntoGraf – Search Functionality

Hence, we can see that the ontology generated using the proposed model gives a very good analysis of the given software project. It represents the key features implemented in the project and also identifies the components and classes that implemented that feature. For developers interested in dependency based analysis of the project, they can visualize the components and features generated using the dependency based approach. The visualization of the generated ontology gives a very clear picture of all the architecture elements recovered using the proposed model: features, components, connectors, interfaces. One more advantage of building an ontology is that ontology is similar to RDF format. Hence the generated ontology can be queried using any RDF querying language like SPARQL.

SPARQL can be used to query the generated ontology and extract required information. Following are some of the useful queries that can be done on the generated ontology:

- 1) Get the list of Features
- 2) Identify components implementing a feature
- 3) Get the list of variants for a feature
- 4) Identify the classes implementing a feature variant
- 5) Get the connectors/dependencies between components

The below figure shows a simple query to fetch all the features in a project using SPARQL language:

```
SPARQL query:
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX base: <http://www.isi.edu/ikcap/geosoft/ontology/software.owl#>
SELECT ?subject
      WHERE (?subject rdfs:subClassOf base:FeatureDomain)
```

```
query
options
cloud
checker
value
resource
sample
regenerator
iterator
update
command
post
common
```

Figure 36: SPARQL [5] query to fetch features in a project

The below figure shows a simple SPARQL query to fetch all the variants of a particular feature that are implemented in a project. The figure also shows the results of the query. Similarly queries can

be written to fetch the list of components, classes and their relationships. Hence the generated model can be queried to search for any functionality and get the components/classes that implement the respective feature.

```
SPARQL query:
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX base: <http://www.isi.edu/ikcap/geosoft/ontology/software.owl#>
SELECT ?subject
WHERE (?subject rdfs:subClassOf base:writer)
```

NaNFloatWriter
BinaryQueryResponseWriter
PythonWriter
PHPWriter
RubyWriter
JSONWriter
XMLWriter

Figure 37: SPARQL [5] Query to fetch variants of a feature

4.1.5 Performance Analysis

One of key goals of the proposed solution is to analyze a given system in a time efficient manner. The proposed solution is built on SPARK framework. Spark is a parallel processing engine, which processes huge volumes of data in less time. Scala and Java programming languages are used for building the proposed solution. The performance of the proposed solution is tested both on a standalone machine and spark cluster with 4 nodes. The size of the Solr jar file is 2.71 MB. Since the data is not huge, there is not much difference in the performance statistics of both the standalone machine and the spark cluster. Interestingly, for programs using sequential processing logic, the performance of single node is better than the multi-node performance. The Java call graph project used for extracting the static calling relationships in the given source code is an open source library. Many

intermediate outputs are generated during the analysis and are used in further processing. Below table shows the major intermediate outputs generated and their sizes.

Table 19: Solr Intermediate Outputs

Intermediate Outputs	Size (MB)
Source Jar	2.71
Static Callgraph Output	8
Class Metadata Documents	0.662
Class Documents after lemmatization	0.777
Training Data for Classification	0.303
Class Dependency Vectors (Kmeans Input)	2.35
Component to Class Mapping	0.04
Feature to Variant Mapping	0.054
Component to Feature Mapping	0.132
Class Variant Mapping	1.36
Connector	0.02
Ontology generated	12.5

The total time taken for generating the ontology for Solr project is 1462606 milliseconds. The below table shows the run time statistics of major steps in the proposed model on single node and 4-node cluster.

Table 20: Run time statistics

Tasks	Single Node Time (milliseconds)	4-Node Time (milliseconds)
Data Extraction	36152	36152
Data Transformation	5829	4679
Machine Learning (Feature based Component detection)	741279	658157
Supervised Learning(Dependency based Component detection)	598965	462785
Connector identification	37638	38592
Ontology Generation	42743	44836
Total	1462606	1245201

From the overall run time statistics, it is clear that the proposed solution is time efficient. It provides both feature based and dependency based analysis within marginal time which is very less compared to the time taken for manual analysis of the project. Analyzing 846 classes and identifying their dependencies and features manually is a very time consuming task. The proposed solution does not require any manual intervention. The proposed solution also provides nearly 87% accuracy in determining the components.

The below graph shows the run time statistics of the machine learning algorithms on a single node and the 4-node cluster. From the graph, it is clear that maximum time is taken by the TF-IDF compared to the rest of the algorithms. This is because, for calculating the TF-IDF values, the algorithm needs to parse all the class documents for each word. Also, once the important features are determined using TF-IDF, there is further processing involved to trace the word back to its full word in the class metadata document.

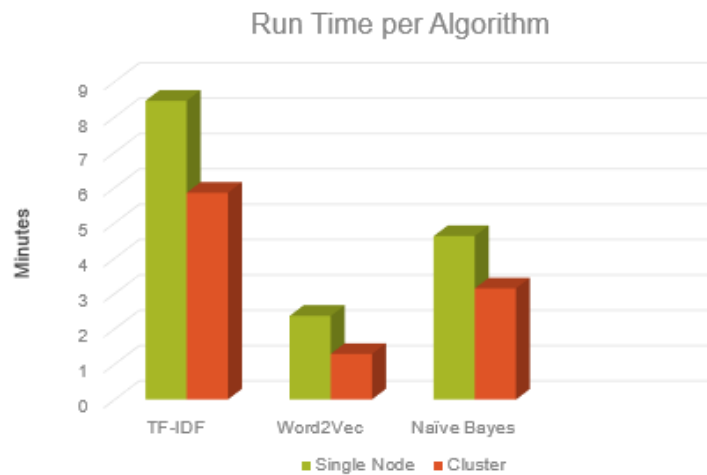


Figure 38: Run time statistics of machine learning algorithms

4.2 Apache Lucene

Apache Lucene is a Java based library for search engines. It is an open source software that provides indexing based on Java and many search features like spell checking, highlighting, etc. Apache Solr is a search server built using Lucene core. Both the projects Apache Solr and Apache Lucene are analyzed separately to identify the components and features. Finally the call graph information generated from Apache Solr project is used to analyze the dependency relationships between the Apache Solr and Apache Lucene projects. Two types of components are detected using the proposed solution: feature based and dependency based. For feature based analysis is based on the assumption that the code entities are named meaningfully and follow camel case naming convention. For dependency analysis, class dependency matrix is built based on the information generated in static call graph output. Below are the statistics of Apache Lucene project generated using Java call graph.

Table 21: Apache Lucene Statistics

Entity	Number
Dependencies	56378
Packages	23
Classes	746
Methods	7984
Parameters	1701

Apache Lucene has 746 classes with many methods and dependencies. Java call graph project is used for identifying all the static dependencies in the Apache Lucene core source jar. Below is the sample output from the call graph:

Table 22: Apache Lucene Sample Callgraph Output

ClassName:MethodName\$parameter	(typeOfCall)ClassName:MethodName\$parameter
C:org.apache.lucene.util.packed.PagedMutable	org.apache.lucene.util.packed.AbstractPagedMutable
M:org.apache.lucene.util.packed.PagedMutable:newMutable	(S)org.apache.lucene.util.packed.PackedInts:getMutable
M:org.apache.lucene.analysis.NumericTokenStream:toString	(M)java.lang.StringBuilder:toString
C:org.apache.lucene.analysis.ReusableStringReader	org.apache.lucene.analysis.ReusableStringReader
M:org.apache.lucene.analysis.Token:<init>	(M)org.apache.lucene.analysis.Token:setOffset
M:org.apache.lucene.analysis.Token:reinit	(O)org.apache.lucene.analysis.Token:copyToWithoutPayloadClone
M:org.apache.lucene.analysis.Token:copyToWithoutPayloadClone	(O)org.apache.lucene.analysis.tokenattributes.PackedTokenAttributeImpl:copyTo
M:org.apache.lucene.analysis.TokenStreamToAutomaton:toAutomaton	(M)org.apache.lucene.analysis.TokenStream:incrementToken
M:org.apache.lucene.analysis.TokenStreamToAutomaton:toAutomaton	(I)org.apache.lucene.analysis.tokenattributes.PositionIncrementAttribute:getPositionIncrement
M:org.apache.lucene.index.CheckIndex:checkIndex	(M)org.apache.lucene.index.SegmentInfo:maxDoc
M:org.apache.lucene.index.CheckIndex:checkIndex	(M)java.lang.StringBuilder:append
M:org.apache.lucene.index.CheckIndex:testTermVectors	(M)org.apache.lucene.index.TermsEnum:postings
M:org.apache.lucene.index.ConcurrentMergeScheduler:setMaxMergesAndThreads	(M)java.lang.StringBuilder:toString
M:org.apache.lucene.index.ConcurrentMergeScheduler:setMaxMergesAndThreads	(O)java.lang.IllegalArgumentException:<init>
M:org.apache.lucene.index.ConcurrentMergeScheduler:setDefaultMaxMergesAndThreads	(S)java.lang.Runtime:getRuntime
M:org.apache.lucene.index.ConcurrentMergeScheduler:setDefaultMaxMergesAndThreads	(M)java.lang.Runtime:availableProcessors
M:org.apache.lucene.index.ConcurrentMergeScheduler:setDefaultMaxMergesAndThreads	(S)java.lang.System:getProperty

Below table shows the call graph statistics for a sample set of classes in Apache Solr. It displays the number of methods, attributes, calling classes, calling methods, etc. for a class.

Table 23: Apache Lucene Sample Callgraph Statistics

Class	Methods	Attributes	Calling Classes	Calling Methods
org.apache.lucene.index.IndexWriter	134	10	66	190
org.apache.lucene.index.FilterLeafReader	57	6	5	52
org.apache.lucene.codecs.lucene50.Lucene50DocValuesProducer	56	21	29	45
org.apache.lucene.util.packed.PackedInts	56	18	21	13
org.apache.lucene.util.fst.FST	54	8	20	47
org.apache.lucene.index.DocumentsWriterFlushControl	51	4	11	31
org.apache.lucene.index.SegmentInfos	48	5	20	64
org.apache.lucene.codecs.compressing.CompressingTermVectorsReader	46	8	28	43
org.apache.lucene.util.automaton.RegExp	44	4	6	17
org.apache.lucene.index.LeafReader	44	3	6	18
org.apache.lucene.util.automaton.Operations	42	5	13	30
org.apache.lucene.search.LRUQueryCache	41	6	18	20
org.apache.lucene.search.LRUFilterCache	40	4	12	19
org.apache.lucene.index.ConcurrentMergeScheduler	38	3	9	15
org.apache.lucene.search.MultiPhraseQuery	36	9	25	39
org.apache.lucene.util.TimSorter	35	1	1	2
org.apache.lucene.codecs.compressing.CompressingStoredFieldsReader	35	11	26	45
org.apache.lucene.util.automaton.Automaton	35	6	7	5
org.apache.lucene.index.SegmentCommitInfo	35	1	4	8
org.apache.lucene.document.FieldType	34	3	3	2
org.apache.lucene.search.SloppyPhraseScorer	34	6	12	23
org.apache.lucene.index.SlowCodecReaderWrapper	34	12	9	23
org.apache.lucene.index.DocumentsWriter	34	6	14	65
org.apache.lucene.index.CheckIndex	34	11	45	111
org.apache.lucene.document.Field	33	20	10	29
org.apache.lucene.index.SegmentReader	33	1	17	22
org.apache.lucene.index.CodecReader	33	4	19	18
org.apache.lucene.util.NumericUtils	31	8	6	11
org.apache.lucene.search.spans.SpanOrQuery	31	5	10	30

4.2.1 Feature Based Analysis

The proposed model performs two types of analysis: semantic analysis and dependency analysis. Semantic analysis generates the feature based components. These components are clusters of classes that implement similar functionality. Similar to Apache Solr, two types of class documents class only and class + dependency are generated for semantic analysis of Apache Lucene. The class documents are then transformed to split each of these names based on camel case and then lemmatized to convert all the words to their base forms. During lemmatization, all the stop words are eliminated. The transformed data is then used to identify features based on word count. Features are basically the important words determined using the word frequency. Below figure shows a sample set of important words in Apache Lucene project.

Word	Count	Word	Count	Word	Count
index	1658	clinit	341	attribute	219
util	1467	write	329	position	212
init	1245	add	320	bit	209
search	922	span	316	document	208
term	863	pack	307	freq	196
field	754	ref	293	score	196
byte	742	sort	283	enum	190
value	733	iterator	275	state	190
reader	575	query	268	scorer	189
store	511	segment	266	leaf	184
codec	406	file	258	automaton	184
info	368	merge	244	overflow	184
next	352	size	241	similarity	184
read	346	ram	235	hash	183

Figure 39: Apache Lucene Important Words

For feature based analysis, the top two words are selected from each class based on word count. The top two words from all the classes become the important words. For each of these words, synonyms are detected using Word2Vec machine learning algorithm and a feature synonym document

is constructed. This document becomes the training data for the supervised learning classification algorithm. Naïve bayes classifier is used for classification. Once the classifier builds a model using the training data, the class documents are used for prediction. The Naïve Bayes algorithm classifies each class to a feature label. All the classes with similar label are grouped together to form a feature based component. Below table shows sample output from Naïve Bayes classifier using class only data.

Table 24: Naïve Bayes Sample Output for Apache Lucene Class Only data

Class	Feature
org.apache.lucene.search.similarities.NormalizationH3	search
org.apache.lucene.search.similarities.NormalizationH2	search
org.apache.lucene.util.DocIdSetBuilder	length
org.apache.lucene.search.similarities.NormalizationH1	search
org.apache.lucene.index.ParallelLeafReader	value
org.apache.lucene.util.ArrayTimSorter	sort
org.apache.lucene.util.automaton.Lev1TParametricDescription	distribution
org.apache.lucene.index.CoalescedUpdates	timestamp
org.apache.lucene.codecs.DocValuesProducer	binary
org.apache.lucene.search.BooleanTopLevelScorers	scorer
org.apache.lucene.document.SortedNumericDocValuesField	field
org.apache.lucene.index.Term	clone
org.apache.lucene.util.FilterIterator	timestamp
org.apache.lucene.util.NumericUtils	equal
org.apache.lucene.search.PhraseQuery	create
org.apache.lucene.codecs.lucene50.Lucene50PostingsReader	enum
org.apache.lucene.util.packed.Direct64	pack
org.apache.lucene.index.MergePolicy	merge
org.apache.lucene.search.QueryCache	search
org.apache.lucene.index.FieldInfos	field
org.apache.lucene.index.MultiPostingsEnum	position
org.apache.lucene.util.Accountable	resource
org.apache.lucene.search.SortedNumericSelector	value
org.apache.lucene.index.FlushPolicy	segment
org.apache.lucene.util.AttributeFactory	attribute
org.apache.lucene.search.similarities.LMSimilarity	search
org.apache.lucene.util.AttributeReflector	keyword
org.apache.lucene.search.NumericRangeQuery	equal

The classification algorithm classifies the classes based on the features. The output of the classification algorithm is a tuple (class, feature). Using this output, classes are grouped together based on the feature identified during classification. Since these components are determined based on features, these set of components are called feature based components. Naïve Bayes classification generated 100 components using Class+Dependency data and 123 components using class only data. Below table shows a sample of feature based components using class only data.

Table 25: Feature Based Components using Class Only

Component	Classes
merge	org.apache.lucene.index.MergePolicy org.apache.lucene.index.MergedPrefixCodedTermsIterator org.apache.lucene.index.SerialMergeScheduler org.apache.lucene.index.LiveIndexWriterConfig org.apache.lucene.index.ConcurrentMergeScheduler org.apache.lucene.search.TopDocs org.apache.lucene.index.IndexWriterConfig org.apache.lucene.index.TieredMergePolicy org.apache.lucene.index.LogMergePolicy org.apache.lucene.index.DocValuesFieldUpdates org.apache.lucene.index.UpgradeIndexMergePolicy org.apache.lucene.index.MergeScheduler org.apache.lucene.index.SegmentInfos org.apache.lucene.index.SegmentMerger
lock	org.apache.lucene.store.FSLockFactory org.apache.lucene.store.SimpleFSLockFactory org.apache.lucene.store.BaseDirectory org.apache.lucene.store.NoLockFactory org.apache.lucene.store.NativeFSLockFactory
write	org.apache.lucene.util.packed.AbstractBlockPackedWriter org.apache.lucene.codecs.lucene50.ForUtil org.apache.lucene.codecs.blocktree.BlockTreeTermsWriter org.apache.lucene.util.IntBlockPool org.apache.lucene.codecs.lucene50.Lucene50SkipWriter org.apache.lucene.codecs.MultiLevelSkipListWriter org.apache.lucene.util.packed.BlockPackedWriter

Following table shows the output of Naïve Bayes classification using class + dependency data.

Table 26: Naïve Bayes Sample Output for Apache Lucene Class+Dependency Data

Class	Feature
org.apache.lucene.index.MultiReader	context
org.apache.lucene.search.similarities.NormalizationH3	similarity
org.apache.lucene.search.similarities.NormalizationH2	similarity
org.apache.lucene.util.DocIdSetBuilder	util
org.apache.lucene.search.similarities.NormalizationH1	similarity
org.apache.lucene.index.ParallelLeafReader	numeric
org.apache.lucene.util.ArrayTimSorter	util
org.apache.lucene.util.automaton.Lev1TParametricDescription	automaton
org.apache.lucene.index.CoalescedUpdates	index
org.apache.lucene.codecs.DocValuesProducer	numeric
org.apache.lucene.search.BooleanTopLevelScorers	search
org.apache.lucene.document.SortedNumericDocValuesField	numeric
org.apache.lucene.index.Term	code
org.apache.lucene.util.FilterIterator	iterator
org.apache.lucene.util.NumericUtils	long
org.apache.lucene.search.PhraseQuery	span
org.apache.lucene.search.FieldDoc	search
org.apache.lucene.util.fst.ReverseBytesReader	util
org.apache.lucene.util.QueryBuilder	span
org.apache.lucene.search.similarities.LambdaTTF	similarity
org.apache.lucene.codecs.lucene50.Lucene50PostingsReader	header
org.apache.lucene.util.packed.Direct64	packed
org.apache.lucene.index.MergePolicy	merge
org.apache.lucene.search.QueryCache	span
org.apache.lucene.index.FieldInfos	norm
org.apache.lucene.codecs.LiveDocsFormat	format
org.apache.lucene.store.IOContext	code
org.apache.lucene.search.payloads.SpanNearPayloadCheckQuery	span
org.apache.lucene.util.packed.BulkOperationPackedSingleBlock	pack
org.apache.lucene.search.Rescorer	collector
org.apache.lucene.util.TimSorter	seek
org.apache.lucene.util.CommandLineUtil	already
org.apache.lucene.search.similarities.Distribution	collector
org.apache.lucene.search.spans.SpanContainQuery	span
org.apache.lucene.search.spans.ConjunctionSpans	span
org.apache.lucene.search.MatchAllDocsQuery	filter

The classes with the same feature label are grouped together to create components. Following table shows sample output of feature based components using class+ dependency data.

Table 27: Feature Based Components Sample Output using Class +Dependency Data

Component	Classes
store	org.apache.lucene.store.Lock org.apache.lucene.store.LockReleaseFailedException org.apache.lucene.store.RAMInputStream org.apache.lucene.store.SimpleFSDirectory org.apache.lucene.store.InputStreamDataInput org.apache.lucene.store.BufferedChecksum org.apache.lucene.store.NIOFSDirectory org.apache.lucene.store.RAMDirectory org.apache.lucene.store.MMapDirectory org.apache.lucene.store.Directory org.apache.lucene.store.LockVerifyServer
token	org.apache.lucene.analysis.TokenFilter org.apache.lucene.analysis.Tokenizer org.apache.lucene.util.StrictStringTokenizer
comparator	org.apache.lucene.search.SimpleFieldComparator org.apache.lucene.search.Sort org.apache.lucene.search.FieldComparatorSource
payload	org.apache.lucene.search.payloads.MaxPayloadFunction org.apache.lucene.search.payloads.MinPayloadFunction
query	org.apache.lucene.search.payloads.SpanPayloadCheckQuery org.apache.lucene.search.spans.SpanNotQuery org.apache.lucene.search.Filter org.apache.lucene.search.spans.SpanFirstQuery org.apache.lucene.search.SearcherFactory org.apache.lucene.search.spans.SpanPositionRangeQuery org.apache.lucene.search.spans.SpanPositionCheckQuery

Components detected using both the inputs are further processed to extract the features specific to each component. The words from all the class documents related to a component are used to create component documents. These component documents are passed to TF-IDF for feature extraction. Once the TF-IDF values are computed for all the words in the component document, the top

three words are selected based on decreasing order of TF-IDF. These words represent the features of the component. Since these words are extracted by splitting the class names, method names etc., the words are traced back to their full name to obtain the feature variant. Also, the class document from which this word is extracted is determined to create the class to feature variant mapping.

Table 28: Apache Lucene Sample Component to Feature Mapping

Component	Feature	Feature Variant
abort	search	lookupFromSearcher
	manager	ReferenceManager
	reference	ReferenceManager swapReference
analysis	attribute	TermToBytesRefAttribute PayloadAttributeImpl PayloadAttribute TypeAttributeImpl TypeAttribute BytesTermAttributeImpl BytesTermAttribute
array	util	MathUtil BitUtil
	sorter	TimSorter ArrayInPlaceMergeSorter

Following table shows the sample class to variant mapping for Apache Lucene using class only data.

Table 29: Sample Class to Variant Mapping

Class	Feature	Feature Variant
org.apache.lucene.index.IndexWriter	merge	mergeInit
org.apache.lucene.index.AbortingException	aborting	AbortingException
org.apache.lucene.index.DocumentsWriter	thread	abortThreadState
org.apache.lucene.util.packed.AbstractBlockPackedWriter	writer	AbstractBlockPackedWriter
org.apache.lucene.codecs.TermVectorsWriter	doc	addAllDocVectors
org.apache.lucene.analysis.CachingTokenFilter	attribute	addAttribute

Once the components and features in the given project are identified, the next step is to detect the connectors. Connectors are basically dependency relationships between the components. If any class in one component is dependent on any class in another component, then a directed relationship is established between both the components. The connectors can be unidirectional or bidirectional. Following table shows sample dependency relationships between various components obtained by using class only data.

Table 30: Sample Connectors in Apache Lucene

Component	Related Components
analysis	attribute
buffer	read file lock
builder	cache pack
directory	file store lock field date
index	merge reader utilsort

4.2.2 Dependency Based Analysis

Callgraph generates the static calling relationships between the classes. These calling relationships are analyzed to detect the dependency based components. There are four types of calling relationships between the classes. Each type of call is assigned a weightage. The degree of dependency between the classes is computed by multiplying the frequency of calls with the weightage. This degree

of dependency is used to construct the class dependency vectors which are then passed to the K-Means [3] clustering algorithm. Below shows the sample dependency vector data for Apache Lucene project.

org.apache.lucene.index.MultiReader	20	20	60	0	0	0	0	0	0	0	0	0	0
org.apache.lucene.search.similarities.NormalizationH3	0	0	0	33.33333	66.66667	0	0	0	0	0	0	0	0
org.apache.lucene.search.similarities.NormalizationH2	0	0	0	33.33333	33.33333	33.33333	0	0	0	0	0	0	0
org.apache.lucene.util.DocIdSetBuilder	25	0	0	0	0	0	12.5	6.25	6.25	6.25	6.25	31.25	12.5
org.apache.lucene.search.similarities.NormalizationH1	0	0	0	50	50	0	0	0	0	0	0	0	0
org.apache.lucene.index.ParallelLeafReader	0	0	0	0	0	0	0	0	0	0	0	0	0
org.apache.lucene.util.ArrayTimSorter	0	0	0	0	0	0	50	0	0	0	0	0	0
org.apache.lucene.search.similarities.LMDirichletSimilarity	0	0	0	0	33.33333	0	0	0	0	0	0	0	0
org.apache.lucene.util.WeakIdentityMap	0	0	0	0	0	0	0	0	0	0	0	0	0
org.apache.lucene.util.automaton.FiniteStringsIterator	0	0	0	0	0	0	4.347826	0	0	0	0	0	0
org.apache.lucene.search.LRUQueryCache	0	0	2.439024	0	0	0	0	0	0	0	0	0	2.439024
org.apache.lucene.search.similarities.LambdaTTF	0	0	0	0	50	0	0	0	0	0	0	0	0
org.apache.lucene.codecs.lucene50.Lucene50PostingsReader	0	0	0	0	0	0	1.219512	0	0	0	0	0	0
org.apache.lucene.util.packed.Direct64	0	0	0	0	0	0	0	0	0	0	0	0	0
org.apache.lucene.index.MergePolicy	0	0	0	0	0	0	0	6.25	0	0	0	0	0
org.apache.lucene.search.QueryCache	0	0	0	0	0	0	0	0	0	0	0	0	0
org.apache.lucene.index.FieldInfos	0	0	0	0	0	0	4.347826	0	0	0	0	0	0

Figure 40: Sample Dependency Vectors for Apache Lucene

K-Means [3] is an unsupervised machine learning algorithm used to cluster the classes into a given no. of clusters(k) . The K-value is determined by using the sum of squared errors technique. The number of clusters is increased until the difference in the sum of squared errors reaches a minimum. For Apache Lucene project, based on the data from degree of dependencies, the optimal K-value is obtained as 112.

Once the K-value is determined, the k value and the dependency data are passed as input to the train method in the K-Means clustering algorithm in Spark MLlib. The algorithm then assigns a cluster number to each class in the input data. The classes are then grouped based on their cluster number to form components. The following table shows a sample output from the K-Means clustering algorithm.

Table 31: Sample Dependency Components for Apache Lucene

Component	Classes
96	org.apache.lucene.analysis.tokenattributes.OffsetAttributeImpl org.apache.lucene.analysis.tokenattributes.KeywordAttributeImpl org.apache.lucene.analysis.tokenattributes.TypeAttributeImpl org.apache.lucene.util.AttributeImpl org.apache.lucene.search.BoostAttributeImpl org.apache.lucene.analysis.tokenattributes.FlagsAttributeImpl org.apache.lucene.analysis.tokenattributes.PayloadAttributeImpl org.apache.lucene.analysis.tokenattributes.PositionIncrementAttributeImpl org.apache.lucene.analysis.tokenattributes.PositionLengthAttributeImpl org.apache.lucene.analysis.tokenattributes.BytesTermAttributeImpl org.apache.lucene.search.MaxNonCompetitiveBoostAttributeImpl
52	org.apache.lucene.util.packed.BulkOperationPacked org.apache.lucene.util.packed.BlockPackedReader org.apache.lucene.util.packed.MonotonicBlockPackedReader org.apache.lucene.codecs.lucene50.ForUtil org.apache.lucene.util.packed.DirectWriter org.apache.lucene.util.packed.AbstractPagedMutable org.apache.lucene.util.packed.GrowableWriter org.apache.lucene.codecs.compressing.CompressingStoredFieldsIndexWriter
4	org.apache.lucene.util.mutable.MutableValueDate org.apache.lucene.util.mutable.MutableValueLong
80	org.apache.lucene.search.spans.SpanPositionQueue org.apache.lucene.search.spans.ContainSpans org.apache.lucene.search.spans.NearSpansUnordered org.apache.lucene.search.spans.SpanWithinQuery org.apache.lucene.search.spans.SpanContainingQuery org.apache.lucene.search.spans.NearSpansOrdered org.apache.lucene.search.spans.SpanScorer org.apache.lucene.search.spans.SpanFirstQuery org.apache.lucene.search.spans.FilterSpans
30	org.apache.lucene.search.spans.ConjunctionSpans org.apache.lucene.search.ConjunctionDISI org.apache.lucene.search.TwoPhaseIterator org.apache.lucene.index.PostingsEnum org.apache.lucene.search.FilteredDocIdSetIterator org.apache.lucene.search.TwoPhaseDocIdSetIterator org.apache.lucene.util.BitSet org.apache.lucene.search.DocIdSet org.apache.lucene.codecs.blocktree.BitSetPostingsEnum
92	org.apache.lucene.search.spans.SpanContainQuery org.apache.lucene.search.spans.SpanNearQuery org.apache.lucene.search.spans.FieldMaskingSpanQuery

4.2.3 Evaluation

Packages are a way of grouping related classes and interfaces by software engineers. For evaluating the components detected using the feature based and dependency based analysis, we have compared the components with the packages and computed the conditional probability using the following formula.

$$P\left(\frac{CompA}{CompB}\right) = \frac{P(CompA \cap CompB)}{P(CompA)} * 100$$

where $P(CompA \cap CompB)$ = Numbers of common classes between the two components

Following is the sample output from the comparison of feature based components generated using class + dependency data and the package clusters.

Table 32: Sample Output for Component to Package Match

Feature Based Component	Package	Percentage Match
pack	org.apache.lucene.util.packed	100
mutable	org.apache.lucene.util.mutable	100
merge	org.apache.lucene.index	92
long	org.apache.lucene.util	50
similarity	org.apache.lucene.search.similarities	93
filter	org.apache.lucene.search	94
store	org.apache.lucene.store	91
file	org.apache.lucene.store	80
collector	org.apache.lucene.search	62
token	org.apache.lucene.analysis	87

Once the percentage matching is computed, the percentage of components which have 100% match, above 70%, above 30% and below 30% is computed. Below pie-charts shows the comparison of components generated from feature based and dependency based analysis with the packages.

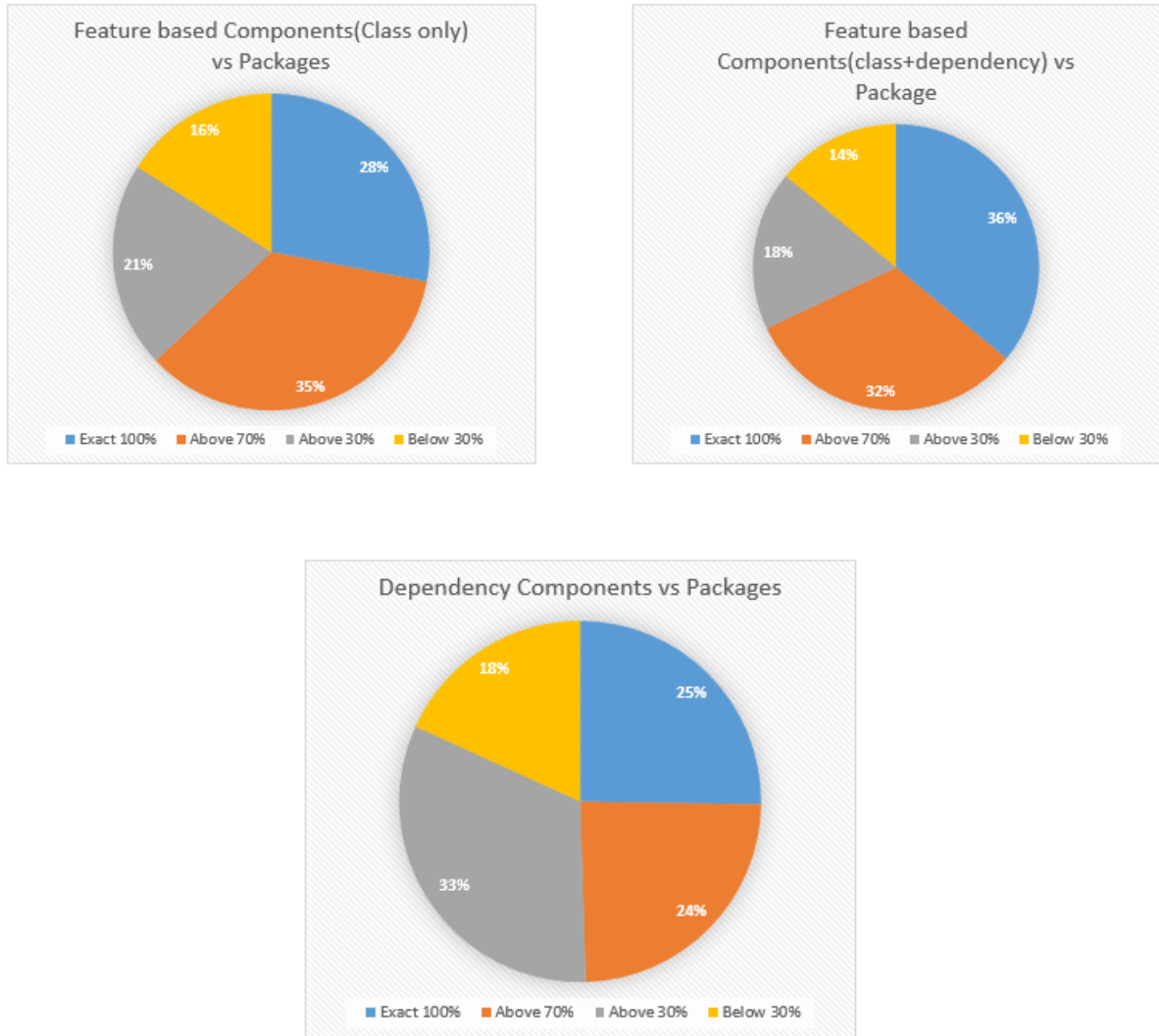


Figure 41: Component Evaluation with Packages

Comparing the results from different approaches, the components generated using feature based analysis with class + Dependency data are more similar to the organization of classes into packages in the source code. This could be because the semantic analysis takes into account the

features implemented in the classes, and the dependency information is included implicitly by adding the information about the calling classes and methods for preparing the class documents.

4.2.4 Performance Analysis for Apache Lucene

The run time statistics of the solution are collected for analyzing the Apache Lucene project similar to Apache Solr. The size of the Apache Lucene source jar is 2.24MB. During the analysis, many intermediate files are created and used for further processing. Below table shows the list of intermediate files and their sizes.

Table 33: Lucene Intermediate Outputs

Intermediate Outputs	Size (MB)
Source Jar	2.24
Static Callgraph Output	8
Class Metadata Documents	0.66
Class Documents after lemmatization	0.25
Training Data for Classification	0.134
Class Dependency Vectors (Kmeans Input)	2.35
Component to Class Mapping	0.034
Feature to Variant Mapping	0.037
Component to Feature Mapping	0.081
Class Variant Mapping	0.812
Connector	0.014
Ontology generated	8.2

The total time taken for performing the feature based and dependency based analysis on the Apache Lucene project and building an ontology for the project is 4158164 milliseconds. The below table shows the run time statistics of major steps in the proposed model on single node.

Table 34: Run time statistics for Apache Lucene

Tasks	Time (milliseconds)
Data Extraction	15504
Data Transformation	4729
Machine Learning (Feature based Component detection)	726367
Supervised Learning(Dependency based Component detection)	498965
Connector identification	7125
Ontology Generation	5474
Total	1258164

4.3 Comparison between Apache Solr and Lucene Components

Apache Solr and Apache Lucene are two open source projects that are analyzed using the proposed model. Below table gives a comparison of the count of various code entities and the time taken for analysis of Solr and Lucene using the proposed model.

Table 35: Apache Solr and Lucene Comparison

Entity	Apache Solr	Apache Lucene
Dependencies	84692	56378
Packages	176	23
Classes	836	746
Methods	8396	7984
Parameters	1812	1701
Source Jar Size (MB)	2.71	2.24
Time for Analysis(milliseconds)	1462606	1258164

The features extracted from Apache Solr and Lucene using the proposed model are compared to find the common features between the two projects. We have discovered 24 common features between the two projects. The below table lists the common features in both the projects.

Table 36: Common Features between Solr and Lucene

S.No	Common Features
1	stream
2	search
3	merge
4	query
5	buffer
6	document
7	directory
8	store
9	token
10	queue
11	block
12	update
13	util
14	index
15	field
16	delete
17	factory
18	char
19	codec
20	similarity
21	ref
22	builder
23	exception
24	writer

Apache Solr is an open source search server built using Lucene Core. Hence there is a dependency relationship between these two open source projects. The static call graph data generated for Apache Solr also captures the dependencies of Solr classes on Apache Lucene code entities. Below is the sample dependency data from the static call graph output of Apache Solr.

Table 37: Sample Dependency Data between Apache Solr and Lucene

Apache Solr	Apache Lucene
M:org.apache.solr.core.SimpleFSDirectoryFactory:create	(O)org.apache.lucene.store.SimpleFSDirectory:<init>
M:org.apache.solr.handler.ReplicationHandler\$DirectoryFileStream:write	(M)org.apache.lucene.store.Directory:openInput
M:org.apache.solr.handler.ReplicationHandler\$DirectoryFileStream:write	(M)org.apache.lucene.store.IndexInput:seek
C:org.apache.solr.handler.ReplicationHandler	org.apache.lucene.index.IndexCommit
C:org.apache.solr.handler.ReplicationHandler	org.apache.lucene.store.Directory
M:org.apache.solr.handler.ReplicationHandler:handleRequestBody	(M)org.apache.lucene.index.IndexCommit:getGeneration
M:org.apache.solr.handler.SnapPuller\$DirectoryFileFetcher:cleanup	(M)org.apache.lucene.store.Directory:deleteFile
C:org.apache.solr.handler.admin.CoreAdminHandler	org.apache.lucene.store.Directory
C:org.apache.solr.handler.admin.CoreAdminHandler	org.apache.lucene.index.DirectoryReader
C:org.apache.solr.handler.admin.LukeRequestHandler\$TopTermQueue	org.apache.lucene.util.PriorityQueue
C:org.apache.solr.handler.component.AbstractStatsValues	org.apache.lucene.queries.function.ValueSource
C:org.apache.solr.handler.component.ExpandComponent	org.apache.lucene.index.AtomicReader
C:org.apache.solr.handler.component.FacetComponent\$DistribFieldFacet	org.apache.lucene.util.FixedBitSet
M:org.apache.solr.handler.component.ShardDoc:<init>	(O)org.apache.lucene.search.FieldDoc:<init>
C:org.apache.solr.handler.component.ShardFieldSorterHitQueue	org.apache.lucene.search.SortField

This dependency data is used to extract the dependency relationship between the components detected in Apache Solr and components detected in Apache Lucene project. If any class in a Solr component is dependent on any class/interface in a Lucene component, then a dependency relationship is established between these two components. Similarly, the dependencies of all the Apache Solr components is detected. The following tables has a sample output of this analysis on

feature based components detected using supervised learning approach. It shows the dependencies of the Solr component “directory” and their respective classes. This component is dependent on three Apache Lucene components: file, buffer, date.

Table 38: Sample Component Dependency between Solr and Lucene

Solr Component	Lucene Component
<p>Component Name: Directory org.apache.solr.core.SimpleFSDirectoryFactory org.apache.solr.core.NIOFSDirectoryFactory org.apache.solr.core.StandardDirectoryFactory org.apache.solr.core.NRTCachingDirectoryFactory org.apache.solr.core.EphemeralDirectoryFactory org.apache.solr.core.MMapDirectoryFactory org.apache.solr.core.RAMDirectoryFactory</p>	<p>Component Name: File org.apache.lucene.search.SearcherLifetimeManager org.apache.lucene.store.FileSwitchDirectory org.apache.lucene.store.RAMDirectory org.apache.lucene.store.Directory org.apache.lucene.index.SegmentInfo org.apache.lucene.store.NRTCachingDirectory org.apache.lucene.store.FSDirectory org.apache.lucene.codecs.lucene50.Lucene50CompoundReader org.apache.lucene.store.LockValidatingDirectoryWrapper org.apache.lucene.store.FilterDirectory org.apache.lucene.store.TrackingDirectoryWrapper Component Name: Buffer org.apache.lucene.store.SimpleFSDirectory org.apache.lucene.store.BufferedChecksum org.apache.lucene.store.NIOFSDirectory Component Name: date org.apache.lucene.store.MMapDirectory org.apache.lucene.index.StandardDirectoryReader org.apache.lucene.index.DirectoryReader</p>

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

This thesis presents a completely automated solution for recovering the architectural elements: Components, Features, Connectors and their relationships and builds an ontology using the information extracted. There is no need for manual intervention. The proposed model generates two types of components: feature based and dependency based. Hence depending on the requirement of the end user, the respective component structure can be analyzed. The features and components of the Apache Solr project detected using the proposed model are compared with the manual architecture recovery results. Above 90% of the features detected in manual analysis are discovered using the automated model. 87.61% of accuracy is attained for detecting components using Naïve Bayes classification algorithm.

This thesis proposes big data analytics solution for analyzing the complex software projects. The data extracted from the software project is generally unstructured or semi structured. Hence machine learning algorithms are used for identifying the hidden similarities in the classes and thereby detecting the components. The proposed solution is built on Spark, and provides a fully automated and scalable model for representing the software. The scalability and performance of the solution has been tested on a single node and on a cluster of nodes. The automated approach is accurate, scalable and time efficient compared to the manual analysis of the project.

This thesis generates an ontology that represents the software system accurately. The features, components, classes, connectors, interfaces and their relationships in the given system are identified and represented in the ontology which can be visualized using tools like Protégé. The proposed solution

also provides a semantic search and query functionality. The generated ontology can also be queried for analysis using querying languages like SPARQL.

The proposed model can be used for automatically analyzing the open source software projects. It can also be used for comparing the different versions of a software project or analyzing the dependencies between related software projects.

5.2 Limitations

The approach presented in this thesis is capable of analyzing only Java based projects. The solution identifies the features based on the semantics. Hence the key assumption is that the code entities are named meaningfully to represent their functionality appropriately. Also, the semantic approach assumes that camel case naming convention is used for naming the code entities.

5.3 Future Work

The proposed model can be extended to identify the provided interfaces of the components. Also the model is based on static callgraph data. The model can be extended using the dynamic call graph information and capture the behavior of the software project. From an implementation stand point, the accuracy of feature detection can be increased using N-gram detection. Also, the proposed model can be extended to software projects written in different languages like Scala, C++, etc. that are object oriented. The performance of the proposed solution needs to be evaluated with big data.

REFERENCES

- [1] GitHub (2014). The Wikipedia [Online]. Available: <http://en.wikipedia.org/wiki/GitHub/>.
- [2] SourceForge (2014). The Wikipedia [Online]. Available: <http://en.wikipedia.org/wiki/SourceForge/>.
- [3] Apache Spark 1.5.1 Mllib. <https://spark.apache.org/docs/1.5.1/mllib-guide.html>
- [4] "The Protege Ontology Editor and Knowledge Acquisition System," <http://protege.stanford.edu/>.
- [5] The SPARQL Query Language for RDF (2008). The W3C [Online]. Available: <http://www.w3.org/TR/rdf-sparql-query/>
- [6] Kang, Kyo C., Cohen, Sholom G., Hess, James A., Novak, William E., Peterson, A. S, Feature-Oriented Domain Analysis (FODA) Feasibility Study, NOV 1990
- [7] Belady, A, L. and Evangelisti, J., C. System partitioning and its measure. *Journal of Systems and Software*, 23, 1981.
- [8] Anquetil, N. and Lethbridge, T. File clustering using naming conventions for legacy systems. In *Proceedings of CASCON*, 1997.
- [9] Garcia, J., Krka, I., Mattmann, C., and Medvidovic, N. 2013. Obtaining ground-truth software architectures. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 901-910.
- [10] Paydar, S., and Kahani, M. 2012. A semantic web based approach for design pattern detection from source code. In *The International Conference on Computer and Knowledge Engineering (ICCKE 2012)*, Mashhad, Iran.
- [11] Binkley, D. 2007. Source code analysis: A road map. In *Future of Software Engineering (FOSE '07)*. 104-119. DOI: 10.1109/FOSE.2007.27
- [12] Sartipi, K. (2003, September). Software architecture recovery based on pattern matching. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on* (pp. 293-296). IEEE.
- [13] RDFCoder – The Java to RDF modeler. 2010. [Online]. <https://code.google.com/p/rdfcoder/>
- [14] The Resource Description Framework (RDF) (2014). The W3C [Online]. Available: <http://www.w3.org/RDF/>.
- [15] Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., & McClosky, D. (2014, June). The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations* (pp. 55-60).

- [16] Regents of the University of California, Java Call graph 1998. <https://github.com/gousiosg/java-callgraph>
- [17] “An Ontology for Software” by Daniel Oberle, Stephan Grimm, Steffen Staab in Handbook on Ontologies, International Handbooks on Information Systems 2009, pp 383-402
- [18] “A probabilistic extension to ontology language OWL” by Zhongli Ding, Maryland University Baltimore, published in System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference.
- [19] Krivov, S., Villa, F., Williams, R., & Wu, X. (2007). On visualization of OWL ontologies. In Semantic Web (pp. 205-221). Springer US.
- [20] Falconer, S. "OntoGraf." Protégé Wiki (2010).
- [21] Ellson, J., Gansner, E., Koutsofios, L., North, S. C., & Woodhull, G. (2002, January). Graphviz—open source graph drawing tools. In Graph Drawing (pp. 483-484). Springer Berlin Heidelberg.
- [22] Software Ontology, Information sciences at University of South California.
<http://www.isi.edu/ikcap/geosoft/ontology/software.owl>
- [23] Horridge, M., & Bechhofer, S. (2011). The OWL API: A Java API for OWL ontologies. Semantic Web, 2(1), 11-21.

VITA

Sravani Punyamurthula completed her Bachelor's degree in Electronics and Communication Engineering from Jawaharlal Nehru Technological University in Hyderabad and then worked as a software development Analyst in Dell International services for 3 years. She then joined Capital One and worked as a principle analyst for a year. Ms. Sravani Punyamurthula started her masters in computer Science at the University of Missouri-Kansas City (UMKC) in August 2014, specializing in Data Sciences and Software Engineering. Upon completion of her requirements for the Master's Program, Ms. Sravani plans to work as a Data Scientist.