

Using Ontology Creation, Text Filtering, and Active Learning to Generate and Optimal Training Set

Joseph M. Geyer

9 May 2009

Abstract

The need to understand software systems is an important part of their update and maintenance. If one does not understand a software system, he/she will have difficulty modifying, maintaining, or updating it. This can be costly in terms of both time and money. Software reverse engineering alleviates this by creating models of a system to aid system comprehension. To understand a software system, it is helpful to decompose it into alternate views. One view is to split the classes of the system into two sets - *domain concept classes* and *non-domain classes*. That is, the classes that relate to the domain of the system, and those classes that just help with the operation and functioning of the system. Supervised learning is a technique that can be used to label *domain concept classes* and *non-domain classes* given a training set. However, manually creating a training set is inefficient. The goal of the proposed research is to present a method and tool to semi-automate the creation of a training set for using supervised learning to classify *domain concept classes* and *non-domain classes* in a software system.

1 Introduction

The documentation of the design of legacy software systems is often neglected, especially as the system ages. It is easy for updates and modifications to be made to a software system without those changes reflected in the design models. Knowing the architecture of a software system is critical to its continued efficacy. The process of regaining this knowledge is termed design recovery. Software reverse engineering (reverse engineering) is important in the maintenance, upkeep, and use of software systems. Maintaining software systems is difficult for many reasons. It is often the case that the designers of a software system are not the same people who will maintain the system [7]. This can cause a gap in the knowledge about the system. The same principle is true for upgrading and using a software system. It is important to fully understand the system in order to best maintain, upgrade, and use it. and Cross [7, page 15] define reverse engineering as “the process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.” There exists an increasing demand for the ability to create models and blueprints of the design of software systems.

Dietterich [8] defines machine learning as “the study of methods for programming computers to learn”. There are many applications for machine learning. Data mining uses machine learning algorithms to find knowledge from data. For example, data mining medical records can lead to medical knowledge [13]. Some other examples are speech recognition, spam classification, autonomous driving, and book recommendation

programs. In each of these examples, the actions of the computer are not explicitly programmed. Learning can be thought of as the process of training a function to correctly give an output based on the previous examples it was given. A binary classification learning problem is where each data example is in one of two distinct classes. Thus, the learning function is being trained to take as input the features of the example and to give as output the classification to which the input features is mapped.

An ontology is a way to better understand a domain. Ontologies consist of concepts in a domain and there interrelationships and hierarchical structure. Algorithms and tools have been developed to reason and draw conclusions in an ontology. Thus, new knowledge is often created via ontologies. Another important use for ontologies is the sharing of information. Since ontologies are built with strict logic rules with hierarchy, users who are interested in a certain domain can often use previously created ontologies related to that topic. The creation of ontologies can either be done manually, semi-automatically, or automatically. Consider the task of creating an ontology from text. Ontology creation from text has its roots in text mining. Often, clustering is used [5, 21, 22, 11, 10] to group words into groups. Hierarchy can be added to these clusters by parsing the text and looking for hierarchical phrasing. For example, the phrase “...*skeleton, bobsleigh, and other winter Olympic sports...*” indicates that *skeleton* and *bobsleigh* are types of *sports*. It also indicates that *olympic sport* is a type of *sport* and that *winter Olympic sport* is a type of *winter sport*. [5] Key concepts from the text can then be readily accessed from the ontology that was created.

A well known problem in the domain of reverse engineering is the concept assignment problem [2]. This is the task of assigning human level concepts to the code that actuates it. Biggerstaff contends that fully understanding a system can be accomplished mapping the concepts to the code that accomplishes those concepts.

In reverse engineering, an alternative view can be useful in understanding a software system. One view is to separate the classes into *domain concept classes* and *non-domain classes*. The *domain concept classes* are related to the domain of the software system. For example, in a banking software system, a class named `account.java` is related to the banking domain, thus it is a *domain concept class*. However, the class `JFrameHandler.java` is not related to the banking domain, rather it helps the system function. It is a *non-domain class*. Carey and Gannod [6] have developed a tool to automate the classification of the *domain concept classes* in a software system by using supervised learning. The training examples are classes in a software system where the input features are object oriented metrics about the class. The output of the training examples is whether the class is a *domain concept class* or a *non-domain class*. This requires a great amount of effort to manually label a training set. The goal of the proposed research is to present a method and tool to semi-automate the creation of a training set for using supervised learning to classify *domain concept classes* and *non-domain classes* in a software system.

The rest of this proposal is organized as follows. Section 2 describes background and related work. Section 3 describes our proposed solution.

2 Background and Related Work

2.1 Background

In the domain of reverse engineering, the concept assignment problem is a classic task for recovering design rationale. Another important technique is that of active learning where the user observes the results of the software tool and indicates whether it was correct or not. Finally, in machine learning supervised learning is used successfully in many cases. We are interested in classification area of machine learning where we can classify an item in one of two different classes.

2.1.1 Software Reverse Engineering and the Concept Assignment Problem

In order to maintain or update a software systems, it is important to understand. Then reverse engineering takes a software system and extracts knowledge about the design to further the understanding of the system. Chikofsky and Cross [7, page 15] define reverse software engineering as the ability to “identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.” With the number and complexity of legacy software systems, there is a real demand for reverse engineering.

Reverse engineering is basically a two step process [4].

- Extracting information, and
- Abstracting information

Abstractions are high level models that help the user better understand the system. Two of the subcategories under reverse engineering are redocumentation and design recovery [4, 7]. Redocumentation is where new models or abstractions are created that just give a different view of the system. No new meaning is given, usually just a more useful perspective. An example of this is printing out the source code in more readable text or creating diagrams directly from source code [7]. Design recovery is more invasive and results in new knowledge about the system. The ultimate goal for design recovery is that the user can have full knowledge of the system.

The famous concept assignment problem presented by Biggerstaff [2] is an example of design recovery that involves two parts. The first is to identify the real human concepts that are in a software system. For example, a concept might be to “calculate revenue”. Then the user assigns that concept (calculate revenue) to the code that accomplishes that concept. This was first described on a line-by-line micro level of the code. An engineer should be able to assign human semantics to segments of code. The ultimate goal of accomplishing this task is to thoroughly understand the software.

A valuable view of a software systems is do divide the classes into the set of *domain concept classes* and *non-domain classes*. The set of *domain concept classes* consist of classes that are associated with the domain of the software system and the the set of *non-domain classes* are those classes that aid in the operation and functioning of the system For example, in a tax calculating software system, a class called *earnings.java* relates to the domain of the system while the class *JPanelAdjuster.java* relates to the user interface and does not have a connection to the domain of personal taxes. In Carey and Gannod’s [6] work, they consider classes as either *domain concept classes* or *non-domain classes*. They assigned each class a vector of object

oriented metrics and used machine learning to classify the *domain concept classes* and *non-domain classes*. They used supervised learning with cross validation to assess the validity of their classifier.

2.1.2 Active Learning

In machine learning, active learning is where the user is involved with the classification task. The learner classifies examples and then presents relatively small homogeneous groups to the user for acceptance or rejection. The groups presented to the user should be of the same type so that the user can quickly identify misfits. It should be small so that the task is manageable. The system then uses a feedback loop to send the user confirmed classifications back to retrain the learner and re-classify all unlabeled examples. Again, this is presented to the user in small homogenous groups. This cycle continues until the user is satisfied with the quality of the classification, or the training set. The training set is continually enlarged and improved. Active learning was successfully used in the the work of Bowring et al. [3] to identify program behavior.

2.1.3 Machine Learning

Machine learning algorithms enable computers to solve problems without being explicitly programmed. The machine changes its behavior to gain improved performance for subsequent iterations of the problem [13]. Machine learning is a specialized field of artificial intelligence and can be broken up into further categories. Unsupervised learning is where no knowledge is given about the output of the training samples. A common solution for this type of problem is to use clustering. Unsupervised learning problems can also be regression or classification. Reinforcement learning is where a sequence of decisions are made with a reward function. An example of this is training a robot to drive a car. There is not a single classification being done for this task, or a single target value to achieve. The goal is a series of acceptable actions for a larger goal. Finally, supervised machine learning involves using training examples that are labeled with the appropriate output. If the output is continuous, it is known as a regression problem. If the output is in the range of a small number of discrete variables, it is a classification problem. For our problem, we will focus on this - classification supervised learning.

We are interested in supervised learning for a binary classification problem. This means that we have a training set with labeled outputs that can be used to train the learner. In a general sense, the problem can be defined in the following way. The variable x is a vector of input variables and y is the output variable and can either be a value from $\{-1, 1\}$. The vector x consists of any number of features depending on the problem. Thus, the feature vector or input vector with n features is

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

The ordered pair $(x^{(i)}, y^{(i)})$ is called a training examples. The training set is composed of m training

examples so that i has a range from 1 to m .

$$trainingset = \begin{bmatrix} (x^{(1)}, y^{(1)}) \\ (x^{(2)}, y^{(2)}) \\ \vdots \\ (x^{(m)}, y^{(m)}) \end{bmatrix}$$

We want to use the training set to learn a function $h : X \rightarrow Y$, called the hypothesis, where X is the space of input variables and Y is the space of output variables. This mapping should accurately classify a sample based on the value of its feature vector. The goal of learning this function is to find the parameter vector θ that can be used to get accurate performance.

One way to understand support vector machines is to visualize the problem geometrically. Each training example in the training set can be placed in a hyper plane of degree n where n is the number of features in the feature vector x . Each example point has output of either +1 or -1 and can be identified with some binary identification scheme, like using a circle for +1 or a triangle for -1. If the two sets of points are linearly separable, a hyper plane can be used to separate the samples into the two classifications. But, there are infinitely many separating hyper planes. The goal then is to find the hyper plane that gives the maximal distance between the closest vectors on either side. These vectors that are closed are called the support vectors. This problem becomes more complicated when the training examples are not linearly separable. However, by projecting these samples into a higher dimensional hyper plane, the points can be linearly separable.

2.2 Related Work

In this section we will discuss some of the other research that is related and similar to this proposal. Sartipi et al. [15, 14] have investigated the use of data mining with reverse software engineering. They created a tool to build alternate views of the structure of a software system by clustering modules. Shirabad et al. [17] considered reverse engineering with data mining by analyzing the relationships of files and code segments in a software system. The problem of semi-automating the creation of a training set for a learner was researched by Tang et al. [18] in the context of video annotations. And finally, approaches for building an ontology from text has been investigated [5, 20, 10]. Caraballo [5] conducted research on text mining to find hierarchical structure of words and concepts.

2.2.1 Software Reverse Engineering through Data Mining

Sartipi et al. [15, 14] have used data mining with reverse software engineering. The view of the structure of a legacy software system is important in understanding it. The goal of their research is to create an improved view of the structure of the software system and a tool to actually restructure the system using an architectural query language. The queries are matched based on clustering using a score function. They define a module in a software system as the set of functions, the set of data types, and the set of variables. These sets are all defined in a similar way. Consider the set of functions in a module. The definition of this set is that they are either contained in the module or they are imported from another module. They can also be exported to other modules. Using clustering, a frequent item set is created based on how often a

function calls another function, uses a data type, or uses a variable. These clusterings are candidates for a module. Finally, these modules create the structural view of the software system.

This research is different from ours for several reasons. The ultimate goal of their research is to create a structural view or possibly restructure a software system where we are ultimately trying to identify classes as either concept or not. But more specifically, the research we are proposing involves the semi-automatic creation of a machine learning training set, where they are using data mining to look at clustering of module based on the source code.

Shirabad et al. [17] also conducted research in reverse software engineering with a data mining approach. They considered the problem of identifying the interdependence of files or routines on a given block of code. In other words, when a user encounters some code, what are the other files or routines that would be relevant or helpful in understanding and maintaining that piece of code? Thus, two files can either be relevant or not relevant to each other. They also considered the classification that they could be potentially relevant. Data was obtained from logging user's normal interactions with the files in a system. If a user interacts with two files in the same session, they are considered to be more related to each other. This and other attributes make up the features in a data sample. Then, the data is classified and compared to the actual classifications.

2.2.2 Ontology creation through Clustering

Clustering is a method used for ontology building from text [5, 21, 22, 11, 10] . Early work involved simpler similarity metrics and algorithms while current work is much more sophisticated. The general idea is to build the ontology bottom up. After identifying concept terms, a similarity metric is used to measure the closeness of terms. Depending on the closeness, the concept terms will be clustered together. These clusters can represent a new concept. The process continues with these new clusters, building a hierarchy.

Caraballo [5] developed an algorithm for automatically creating a hypernym-labeled noun hierarchy from text corpus. Using the Wall Street Journal, she identified 50,000 nouns. When two or more nouns are used together in a conjunction or an appositive, the assumption is that they are semantically related. Each noun is given a vector where the values are the number of times each other noun is found in either a conjunction or an appositive in the text. Similarity between nouns is then calculated by comparing the angle between each pair of vectors. The most similar nouns are clustered into a new parent node, and the process continues. Hierarchy is then extracted by looking for the use of the word *other* with nouns in a conjunction clause. For example, *birds, squirrels, and other small mammals* indicates that mammal is a hypernym of birds and squirrels. This algorithm correctly assigned hyponyms to randomly selected hypernyms with a 33% accuracy according to human judges participating in the experiment. One of the top three hypernyms that the algorithm produced for a noun matched the judge's hypernym with a 60% accuracy.

Instead of a bottom up approach, Yang and Callan [20] proposed an incremental approach where each term is considered and placed in the correct spot in the hierarchy.

3 Proposed Research

This section discusses the proposed research. It starts by defining the research problem and the proposed solution. Next, it examines the feasibility and validity measures of the proposed solution. This section also

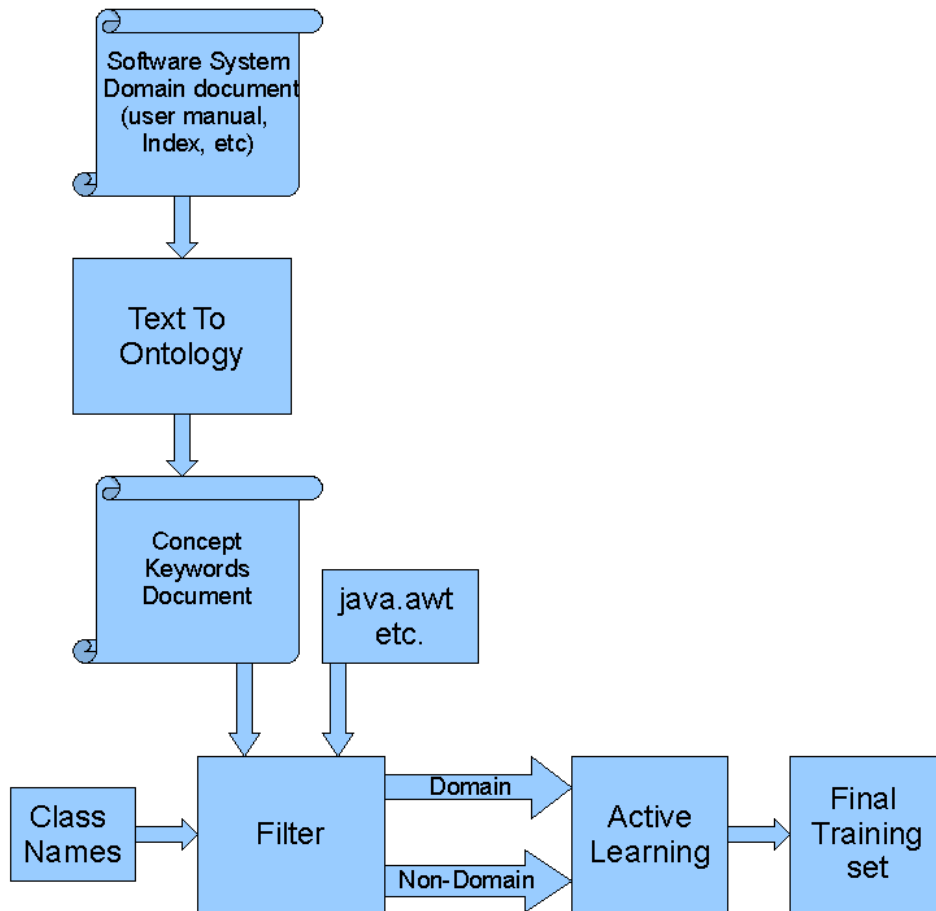


Figure 1: Flowchart Model of Proposed Research

includes some preliminary results conducted on a system in the UML Modeling domain. Finally, a timetable for completing the research is presented.

3.1 Specific Research Problem

Carey and Gannod [6] have developed a method for using supervised learning to separate a software system into *domain concept classes* and *non-domain classes*. The method uses supervised learning where classes of the software system are examples. Each class has an input vector of object oriented metrics that describe it. To train the learner, a training set is used that consists of a set of training examples with input vectors and the corresponding classification of either *domain concept classes* or *non-domain classes*. The learner can then be used on examples that are not yet classified.

One obstacle to this method is the creation of the training set. The user must manually classify a subset of the data so that the learner can automatically classify the rest of the instances in the data set. As an example, if a researcher wants to create a training set that is 10% of a software system with 5000 classes,

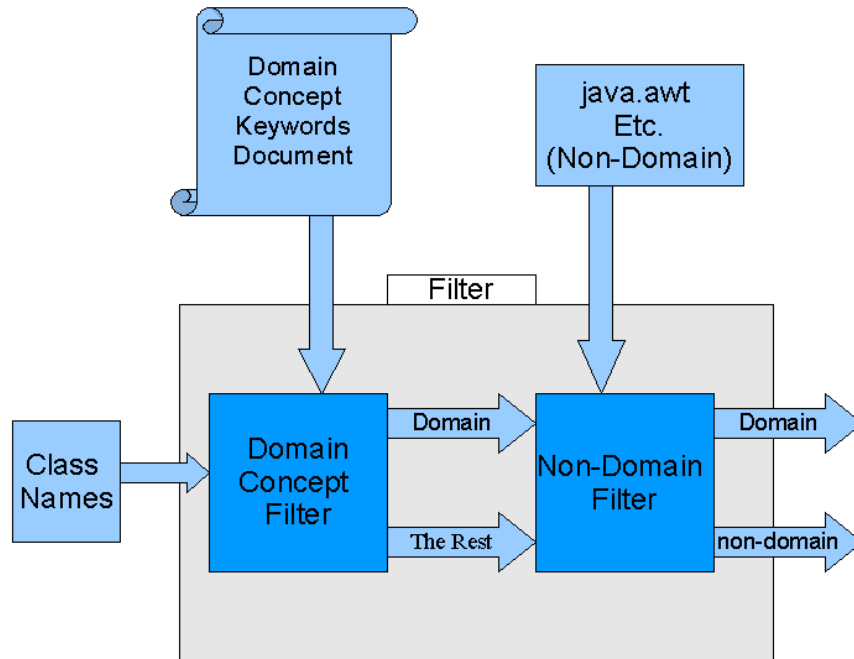


Figure 2: Model of Filter

he/she would have to manually classify 500 classes. This creates a bottleneck in the productivity of the reverse engineering process. Through this research, we want to reduce the load of manually classifying the training set.

3.2 Proposed Solution and Methodology

The flowchart in figure 1 shows a model of the proposed solution. A central piece of our solution is the filter that will positively identify *domain concept classes* and *non-domain classes* through string comparisons. The filter outputs the *domain concept classes* and *non-domain classes* to the active learning stage where the user accepts or rejects the determination of the filter on each class. As part of the active learning stage, the user confirmed training set is then used to train the SVM learner and the whole system is classified. Next, a subset of the classified system is looped back to be accepted or rejected by the user. This feed-back loop continues until the user is satisfied with the quality of the training set, and it is then used to train the learner.

As shown in figure 1, there are two points of input for our system - the *software system domain document* and the *class names*. The *software system domain document* is a text document that is related to the domain of the software system and the *class names* are the actual names of the classes of the software system. From here, our system is automated until the active learning stage, where the user can accept or reject examples for the training set.

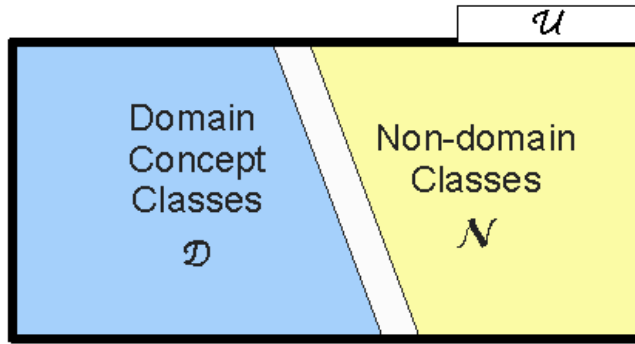


Figure 3: Set Diagram of the Domain and Non-domain classes

3.2.1 Text to Ontology

Software systems have a concept domain associated with its high level functioning. Some domain examples are database, IDE, data mining, UML modeling, chat client, text editor, online store, and project manager. Consider a software system used for UML development. In UML, some examples of conceptual terms are *inheritance*, *use case*, and *association*. We would expect that some of the classes in this system would use these words as part of the class name or as the whole class names. These terms can be taken directly out of a textbook or other document on UML. The same is true of other domains as well. We can extract these terms from any document (user's guides, manuals, textbooks, tutorials, etc.) associated with the domain of the system. We will use this document as input into the text-to-ontology builder (see figure 1. This will create an ontology of all the words in the document. We are especially interested in the clustering and hierarchical structure of this ontology. The ontology will group concepts into clusters and hierarchies that we can use to build our concept filter. Specifically, the set of these keywords will make up the Concept Keywords Document as shown in figure 2.

3.2.2 Training Set Selection via Filtering

The filtering stage consists of a domain concept filter and a non-domain filter as shown in figure 2. In our problem, every class is either a *domain concept class* or *non-domain class*. Both filters works by comparing a class name with the set of strings in the filter. If a match is found, the class name is placed into either the *domain concept class* set or the *non-domain class* depending on the filter.

This method can be described formally as follows. See figure 3

Let \mathcal{U} be the set of all classes, or examples, in the software system.

Let \mathcal{D} be the set of all *domain concept class* examples.

Let \mathcal{N} be the set of all *non-domain class* examples.

where \mathcal{D} and \mathcal{N} are disjoint.

Let \mathcal{T} be the set of classes that are neither *domain concept class* examples nor *non-domain class* examples. (This is the empty set)

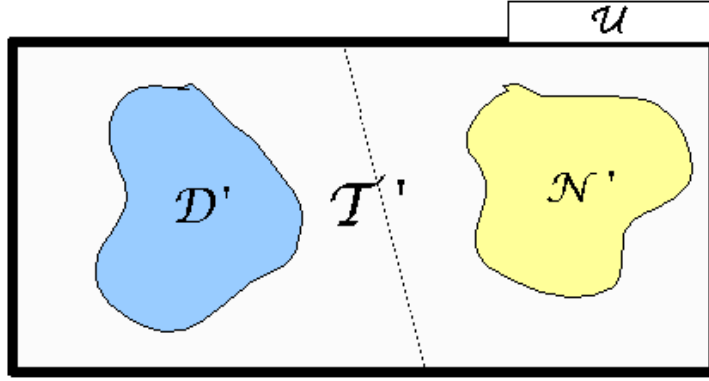


Figure 4: Set Diagram of \mathcal{D}' and \mathcal{N}'

and

$$\mathcal{D} \subset \mathcal{U}$$

$$\mathcal{N} \subset \mathcal{U}$$

$$\mathcal{T} \equiv \mathcal{U} \setminus (\mathcal{D} \cup \mathcal{N}) = \emptyset$$

Then our system is generating a set $\mathcal{D}' \subseteq \mathcal{D}$ and $\mathcal{N}' \subseteq \mathcal{N}$ such that $\mathcal{D}' \cup \mathcal{N}'$ is the training set. See figure 4. Let $\mathcal{T}' \equiv \mathcal{U} \setminus (\mathcal{D}' \cup \mathcal{N}')$ be the set of all class examples that are not yet placed in either \mathcal{D}' or \mathcal{N}' .

Thus, filtering of the classes takes place by examining matches of substrings of the software systems class names with substrings elements of the *domain filter* and the *non-domain filter* as shown in figure 2. If a match occurs on the *domain filter*, then the class name is placed in a list of probable concept classes, \mathcal{D}' . If a match occurs on the *non-domain filter*, then the class name is placed in a list of probable non-domain classes, \mathcal{N}' . If a class does not get matched by either filter, then it is not placed in either list. It is left in the set \mathcal{T}' . In this way, we have a starting training set to build our optimal training set.

3.2.3 Training Set Selection via Active Learning

After the filtering stage, \mathcal{D}' and \mathcal{N}' may have too few elements and/or the elements may be incorrectly placed. Since supervised learning depends on the quality and size of the training set for accurate classification, we will use active learning to help build an optimal training set. For each $d \in \mathcal{D}'$ and for each $n \in \mathcal{N}'$ placed by the filter, the user will be able to accept or reject this placement. This is shown in the User Confirmation (accept/reject) box in figure 5. If the user accepts it, then nothing changes with that element. Otherwise, if the user knows it belongs to the other set, he/she can place it there or if the user is unsure of it's classification, it is placed in \mathcal{T}' shown in figure 4. Once the user is satisfied with each $d \in \mathcal{D}'$ and $n \in \mathcal{N}'$, then $\mathcal{D}' \cup \mathcal{N}'$ is used as the training set for the learner as shown in the "Training set" rectangle in figure 5. After the software system is classified, each element in \mathcal{T}' will be machine labeled. We will randomly select $\mathcal{T}'' \subset \mathcal{T}'$ to feed back to the user for confirmation of the machine classifications. The cardinality of \mathcal{T}'' is between 20 and 50 (to keep the task manageable). This feedback loop is shown in figure 5 with the arrow from "Learner

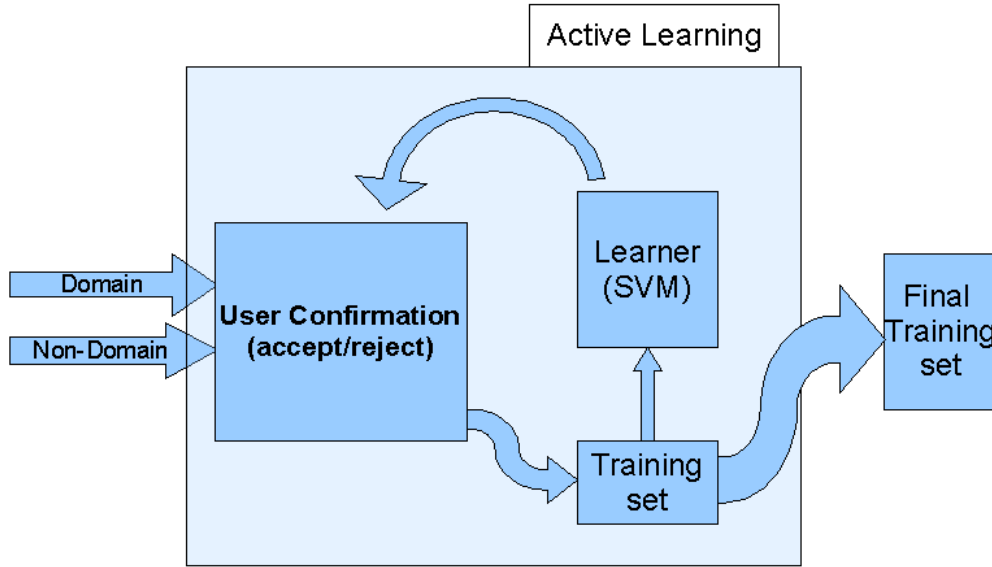


Figure 5: Active Learning Module

(SVM)” back to “User Confirmation (accept/reject)”. If the user agrees with the classification, then the element will be placed in the appropriate set (i.e. \mathcal{D}' or \mathcal{N}') as shown in figure 4. Otherwise, it is placed in the correct set or back into \mathcal{T}' if the user is unsure. This looping process continues until the user is satisfied with the size and quality of the training set. Every iteration of the active learning process creates a larger and more accurate training set.

3.3 Feasibility

The proposed solution will be implemented in an eclipse plugin that utilizes filtering and active learning techniques. This plugin will be an extension on the plugin by Carey et al. [6] We will expand on the support vector machine annotation classification software developed by Carey [6] and expect this to be available. Our tool will be written as a plugin using Eclipse. Data will be acquired from open source software systems like ArgoUML. We will manually label the classes in our experiment and use this as the oracle for our research.

These keywords together make up the *domain concept filter*. That is, these are the terms to which the class names will be compared. Likewise there are other terms that are usually associated with the inner workings of a software system. We compiled a list of 98 terms that would filter out classes that are probably not concept classes. These terms are from the java.awt package. Some examples are *JButtonItem*, *Scrollbar*, and *JTextPane*. These non-concept keywords are used in the *non-concept filter*.

Figure 5 shows the domain and non-domain classes as input into the active learning module. Here the user will be presented with the list of probable domain concept classes generated by the *concept filter*. The user accepts individual classes as concept by using a checkbox. The user will also have checkbox options to re-classify the class or to indicate that he/she cannot confirm the classification. In accepting a class

as a domain concept class, the user is finalizing the positive examples for the training set for the learner. Similarly, the user is finalizing the negative training example by accepting the classification of a non-domain class. This first time through the active learning stage creates an initial training set that is used to train the learner and classify the entire system. Next the user indicates the number, m , of classified examples he/she wants to confirm. The system randomly selects $m/2$ positively classified examples and then $m/2$ negatively classified examples to present to the user for validation. The user is only presented with one set at a time. To improve efficiency and ease to the user, $20 \leq m \leq 50$ and the set of $m/2$ positive and $m/2$ negative examples are presented in two separate steps. The user chooses to end the active learning stage when the training set is acceptable.

Lastly, the final training set is used to train the learner. The learner can classify the whole software system - identifying classes as either *domain concept classes* or *non-domain classes*.

3.4 Validation Approach and Measures

Gueheneuc et al. [9] have developed a framework for evaluating design recovery tools. This framework consists of fifty-three criteria in eight different categories. These categories include the “the *Context* in which it is applied, its *Intent*, its *Users*, its *Input* and *Output*, the *Technique* which it implements, its actual *Implementation*, and the *Tool* itself.” We will use this framework, and the fifty-three criteria to help guide and evaluate the design and goals of our design recover tools. This will help ensure that our tool meets a specific need in the reverse software engineering community.

We will also assess the validity of this approach by calculating the accuracy of the classifications called the classification rate. We need to consider the following cases related to accuracy: classifying true when it is false, classifying false when it is true, classifying true when it is true, and classifying false when it is false. The last two are our goal, but the first two are errors. We would rather not classify an example rather than get it wrong for the training set. This should be accomplished through the active learning process. But we also want to see how an improved training set improves the performance of the classifier.

Finally, we will assess the success of this solution by measuring its efficiency. We will compare a semi-automated training set generator solution to manually labeling the training set. We can look at both the time it takes and the resulting accuracy.

3.5 Preliminary Results

ArgoUML [19] is a Java based open source software engineering tool with 1711 classes. Many of the class names are related to the domain of uml modelling. Some examples of *domain concept classes* are `useCaseDiagram.java` and `stateDiagram.java`. The system also has many *non-domain classes* such as `buttonActionNewSignalEvent.java` and `initDiagramAppearanceUI`. This system is partitioned into many packages, and we decided to experiment on the `diagram` package. With only 300 classes, this was more feasible to manually label a training set to use as a basis for our experimentation. Of these, we manually labelled 130 as *domain concept classes* and 170 as *non-domain classes*.

We used a UML glossary as the domain document to train the domain filter. Parsing through this document, we extracted 206 possible domain concept words to be used to train the domain concept filter. We did not use a text to ontology tool at this point, but rather used these terms directly as a filter. We

hope to get terms more closely related to the domain of the system when we employ the ontology building stage. We also used the `java.awt` class names for our non-domain filter. Since `JButton.java` is a subclass of `java.awt`, this will filter out the class `buttonActionNewSignalEvent.java` because of the match on the string `button`.

The filtering stage produced 123 candidate *domain concept classes* and 88 candidate *non-domain classes* leaving 89 classes without a prediction. Next is the active learning stage. We selected 22 of the 123 *domain concept classes* candidates and 22 of the 88 *non-domain classes* that we thought were accurately filtered. We used these 44 classes as the training set for the SVM learner to label the `diagram` package of 300 classes. For the preliminary experiments, we did not cycle through the active learning stage. The results of this experiment were the following.

Table 1: Results of Experimentation on `diagram` package

		Actual	
		Domain	Non-Domain
Predicted	Domain	115	90
	Non-Domain	14	81

The correct *domain concept class* predictions and the correct *non-domain class* predictions are the values 115 and 81 respectively. A false negative, where we predicted a class to be a *non-domain class* but it was actually a *domain concept class*, occurred 14 times. Finally, a false positive, where we predicted a class to be a *domain concept class* but it was actually a *non-domain class* occurred 90 times. Thus, the accuracy of the system shown in table 1 can be calculated as the number of predictions that were accurate divided by the total number of classes.

$$\begin{aligned}
 Accuracy &= \frac{115 + 81}{115 + 90 + 14 + 81} \\
 &= 0.653
 \end{aligned}$$

3.6 Timetable

Date	event
May 19	Thesis Proposal written
August 30	Thesis Proposal
TBA	Implement tool
TBA	Evaluate tool
TBA	Experiment and collect results
TBA	Analyze results
TBA	Thesis Draft
TBA	Final Thesis Draft written
TBA	Defend Thesis

4 Conclusion

We have presented a method and tool to semi-automate the creation of a training set for using supervised learning to classify *domain concept classes* and *non-domain classes* in a software system. This method uses text to ontology creation and string filtering to generate the initial training set. Finally, the training set is optimized through active learning.

With this method and tool, the software system will be deconstructed into a helpful alternative view. This view consists of grouping the *domain concept classes* together and the *non-domain classes* together. This can be a useful step in understanding, maintaining, or updating a software system.

References

- [1] Kristin P. Bennett and Colin Campbell. Support vector machines: hype or hallelujah? *SIGKDD Explor. Newsl.*, 2(2):1–13, 2000.
- [2] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Commun. ACM*, 37(5):72–82, 1994.
- [3] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of software behavior. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 195–205, New York, NY, USA, 2004. ACM.
- [4] Gerardo CanforaHarman and Massimiliano Di Penta. New frontiers of reverse engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society.
- [5] Sharon A. Caraballo. Automatic construction of a hypernym-labeled noun hierarchy from text. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 120–126, Morristown, NJ, USA, 1999. Association for Computational Linguistics.
- [6] M.M. Carey and G.C. Gannod. Recovering concepts from source code with automated concept identification. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 27–36, June 2007.
- [7] E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, Jan 1990.
- [8] T. G. Dietterich. Machine learning. In *Nature Encyclopedia of Cognitive Science*. Macmillan, 2003.
- [9] Y.-G. Gueheneuc, K. Mens, and R. Wuyts. A comparative framework for design recovery tools. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp.–134, March 2006.
- [10] He Hu and Da-You Liu. Learning owl ontologies from free texts. In *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, volume 2, pages 1233–1237 vol.2, Aug. 2004.

- [11] Hyunjang Kong, Myunggwon Hwang, and Pankoo Kim. Design of the automatic ontology building system about the specific domain knowledge. In *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*, volume 2, pages 4 pp.–1408, Feb. 2006.
- [12] Alexander Maedche, Er Maedche, and Steffen Staab. The text-to-onto ontology learning environment. In *Software Demonstration at ICCS-2000 - Eight International Conference on Conceptual Structures, 2000*.
- [13] Mitchell. *Machine Learning*. McGraw-Hill Education (ISE Editions), October 1997.
- [14] K. Sartipi. Software architecture recovery based on pattern matching. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 293–296, Sept. 2003.
- [15] K. Sartipi, K. Kontogiannis, and F. Mavaddat. Architectural design recovery using data mining techniques. In *Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European*, pages 129–139, Feb 2000.
- [16] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of java software. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 10–16, New York, NY, USA, 1998. ACM.
- [17] Jelber Sayyad Shirabad, Timothy C. Lethbridge, and Stan Matwin. Supporting maintenance of legacy software with data mining techniques. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 11. IBM Press, 2000.
- [18] Jinhui Tang, Yan Song, Xian-Sheng Hua, Tao Mei, and Xiuqing Wu. To construct optimal training set for video annotation. In *MULTIMEDIA '06: Proceedings of the 14th annual ACM international conference on Multimedia*, pages 89–92, New York, NY, USA, 2006. ACM.
- [19] Tigris.org. Argouml. URL: <http://argouml.tigris.org/>.
- [20] Hui Yang and Jamie Callan. Metric-based ontology learning. In *ONISW '08: Proceeding of the 2nd international workshop on Ontologies and nformation systems for the semantic web*, pages 1–8, New York, NY, USA, 2008. ACM.
- [21] Hui Yang and Jamie Callan. Ontology generation for large email collections. In *dg.o '08: Proceedings of the 2008 international conference on Digital government research*, pages 254–261. Digital Government Society of North America, 2008.
- [22] Jingtao Zhou, Mingwei Wang, Han Zhao, Shusheng Zhang, and Chao Zhang. Concept capture based on column matching and clustering. In *Semantics, Knowledge and Grid, 2005. SKG '05. First International Conference on*, pages 71–71, Nov. 2005.