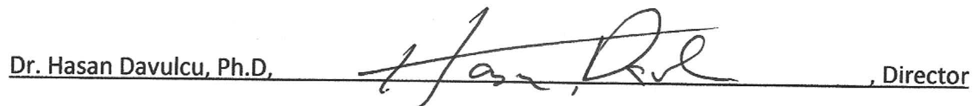
GCKENGINE: AN ALGORITHM FOR AUTOMATIC ONTOLOGY BUILDING

By

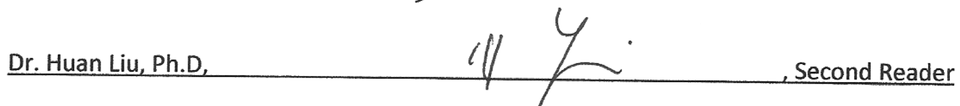
Garret Walliman

Has been approved
March, 2013

APPROVED (printed name, signature):



Dr. Hasan Davulcu, Ph.D, ____________________, Director



Dr. Huan Liu, Ph.D, ____________________, Second Reader

Dr. Rida Bazzi, Ph.D, ____________________, Third Reader

ACCEPTED:

____
Dean, Barrett, the Honors College

0. Abstract

To facilitate the development of the Semantic Web, we propose in this thesis a general automatic ontology-building algorithm which, given a pool of potential terms and a set of relationships to include in the ontology, can utilize information gathered from Google queries to build a full ontology for a certain domain. We utilized this ontology-building algorithm as part of a larger system to tag computer tutorials for three systems with different kinds of metadata, and index the tagged documents into a search engine. Our evaluation of the resultant search engine indicates that our automatic ontology-building algorithm is able to build relatively good-quality ontologies and utilize this ontology to effectively apply metadata to documents.

Acknowledgements:

Much of the early work done on the web crawler and classifier was researched and written by

Forrest Falk (ffalk@asu.edu)

Table of Contents

0. Abstract 1
1. Introduction 4
    1.1. History of the World Wide Web 4
    1.2. The Semantic Web 5
    1.3. Thesis Outline and Summary 7
2. Literature Review 9
    2.1. History of the Semantic Web 9
    2.2. Semantic Web Implementations - Hand-Built Ontology 10
    2.3. Semantic Web Implementations - Automatically-Built Ontology 12
3. Implementation and Architecture 14
    3.1. Main File 15
    3.2. Web Crawler and Hub Extractor 16
    3.3. Classifier and Hand Classifier 17
    3.4. Term Extractor 20
    3.5. Ontology Builder 21
        3.5.1. The googleRelationships Function 22
        3.5.2. The buildRelations Function 24
    3.6. Ontology Classifier 28
    3.7. HTML Tagger 29
    3.8. Solr Engine 34
4. Performance Evaluation 35
    4.1. Tutorial Classification Evaluation 36
    4.2. Ontology Builder Evaluation 40
    4.3. Precision and Recall of the GCKEngine Search Engine 54
5. Analysis 57
6. Future Work 63
7. Conclusion 65
References 67

Introduction

1.1. History of the World Wide Web

The World Wide Web was created, in the words of Tim Berners-Lee, as “a common information space in which we communicate by sharing information” (*Personal History*). His envisioning of the Web is one in which all posted information may be shared and connected together into one great corpus of knowledge. Implicit in his vision is the requirement that this information be highly structured and organized, and that there must exist some method to effectively search and retrieve desired information from the corpus. For much of the history of the Web to date, however, the only way to achieve this organization and connectedness was for individuals to manually structure their online information, to manually insert hypertext links into content to connect them to other pieces of information. This manual structuring is slow, imprecise, and is necessarily limited; the organization of the content is not standardized within a greater Web context, and while an individual may build connections between their own documents and other documents posted on the Web, most individuals have no way of facilitating discovery of their own documents: they cannot insert links into *other* user’s data in order to connect back to their own documents. To solve the problem of document discovery, search engines were developed. Through the use of textual analysis, these search engines are able to organize and retrieve documents via keyword, and to some extent are able to establish relationships between documents based on matching textual patterns. However, this is a low level relationship at best. When searching for a document, it is often difficult to capture exactly what we are trying to find in the form of keyword search. Furthermore, the use of keywords presupposes that we have precise knowledge about the topic we are searching for to begin with. This, along with a host of other problems, clearly limits our ability to search for information with modern search engines.

1.2. The Semantic Web

These problems – the difficulty of effectively organizing, building connections between, searching and retrieving the information hosted on the web – have been recognized for almost as long as the Web has existed. To attempt to solve these problems, the concept of the *Semantic Web* was developed. The Semantic Web proposes, at a high level, that computers should be able to *understand* the information that they are processing. If this can be achieved, then it will be possible to automate the organization and connection of information on the Web – collections of information could be parsed, understood, stored in some organizational standard, and connections between pertinent documents could be automatically applied. On the Semantic Web, searching and information retrieval would be vastly superior to modern keyword-based search engines: a semantic search engine would understand not just the keywords one is searching for, but the relationship *between* these keywords. Such an engine would understand a human-language query entered into it, and could return the most pertinent information to the user based on that meaning. The Semantic Web, then, would achieve the vision Berners-Lee was pursuing in the creation of the World Wide Web.

The actual creation of a Web that could rightfully be called "semantic" is a mammoth task, one that we are only now beginning. However, we have begun taking the first steps towards the realization of this goal. One method that has successfully achieved in many cases some degree of semantic computing is to develop an ontology for a certain corpus of knowledge, and to organize information with this ontology through the application of meta-information to documents. This particular strategy has led to much work over the past decade, as will be detailed in section 2 below. Much of the work accomplished thus far has attempted to automate (or at least make easier) the application of metadata to documents – for example, automatic tagging of documents, or naive information extraction and classification. Automated metadata

application is without a doubt a highly important development, as manual tagging is a tedious and imprecise task. If we can automate metadata application, then we can surely increase the consistency (and hopefully the accuracy) of the metadata. However, for efforts in automated tagging to succeed, an automated tagging algorithm requires an ontology: we must be able to *understand* the information in a document in order to automatically tag it, and to extract and classify information we must know what information is important and what the potential classifications for such data are. Thus the development of good ontologies is a highly crucial task core to all semantic computing efforts. Ontology generation is extremely complex, and because of its complexity it would seem that there have been comparatively fewer attempts at automating ontology creation than there have been at automating document tagging. Perhaps the central challenge to automated ontology creation is the fact that in order to create an effective ontology requires one to understand many high-level language concepts: we must be able to understand the rules of language, as well as understand which of those rules are pertinent to the ontology we are building. Additionally, automatic ontology-building requires access to the actual terms we wish to include in said ontology, organized in some manner that the computer can understand; this is a nontrivial requirement. It is for these reasons that most ontologies used in semantic computing efforts are built partially or entirely by hand.

However, building an ontology by hand is a very onerous task. By its nature, an ontology should be relatively exhaustive within a certain domain, and for it to be truly effective no important terms or concepts may be excluded, and no relationships may be overlooked. Such an exhaustive, comprehensive task would be very well suited to automation if the theoretical and practical difficulties associated with it could be overcome.

1.3. Thesis Outline and Summary

In this thesis, we attempt to take the first steps towards automatic ontology building. Specifically, we gather a large pool of terms which may or may not be related to a certain domain, and use a "relationship-building" algorithm to determine whether the terms in the pool are indeed related at all to the domain. If they are, we attempt to classify them into different kinds of relationships. In this way, an ontology may automatically be built.

We are able to overcome the high difficulty of automatic ontology generation by simplifying the problem in two ways. First, we avoid the problem of the machine understanding which language rules are pertinent to our ontology by predefining the rules and relationships we are seeking. Second, instead of attempting teach the machine how to understand those pertinent language rules, we simply define a system that will allow the machine to recognize correct rule usage, without actually understanding what those rules are. We are able to refer to an external body of knowledge – in this case, the Google search engine – to query two terms which may feature a relationship, and if we can determine whether that relationship is indeed present or not, we can classify those terms accordingly without having to actually understand the relationship itself. While Google, like all modern search engines, is keyword based and thus would not be as good as a semantic engine for true information retrieval, it *is* useful for gathering terms which *could* be part of a domain, as well as determining the degree of relation between terms. Google is particularly useful for this kind of task because it (theoretically) can retrieve documents for every possible domain of knowledge; thus, for any two terms, if a relationship does exist between them, Google should be able to provide us examples of that relationship. The exact way that the algorithm works will be described later, in Section 3.

To show the validity of our automatic ontology building algorithm, we have run the algorithm on a few domains – namely, tutorials for three different computer programming languages or

systems. We attempt to build ontologies for each system (the system here being our domain) consisting of three categories of relationships between the terms and the domains:

1. *Component*: the term is a discrete part of the system – for example, a certain module, function, implementation, technique, or concept.
2. *Version*: the term is a version of the system. Note that while components may also have versions, if the component is a module or implementation, we are only here interested in system versions.
3. *Provider*: the term is the name of a provider of either the system or tutorials for the system – in general, the name of a website or company that is heavily involved with the system.

More details about the nature of these relationship categories will be discussed in Section 4. After building ontologies for these domains, we have tagged and classified tutorial documents with this ontology, and stored the results into an Apache Solr search engine, which we have named the “GCKEngine” or G Computer Knowledge Engine. The various aspects that go into retrieving the tutorials for this engine, classifying them, and tagging them will also be discussed in Section 3.

To determine whether our automatic ontology-building algorithm was successful, we performed multiple tests on each system indexed into the engine. The descriptions of these tests will be discussed in Section 4 and analyzed in Section 5.

Section 6 will discuss the limitations of the basic ontology-building algorithm developed in this thesis, and various future developments that may improve its capabilities. Finally, Section 7 will conclude the thesis.

2. Literature Review

2.1. History of the Semantic Web

The unified Semantic Web concept is generally considered to have begun with an article written by Tim Berners-Lee and colleagues entitled, appropriately, “The Semantic Web” (*Semantic Web 2001*). The article lays out the basic format of how the Semantic Web is intended to work, the main idea of which is that by applying certain metadata to a document, interested systems can analyze that document and gain an understanding of that document’s information. Berners-Lee notes that all Semantic rules used to parse this metadata can take the following form: “particular things (people, Web pages or whatever) have properties (such as ‘is a sister of,’ ‘is the author of’) with certain values (another person, another Web page)” (*Semantic Web 2001*). These rules will be used by programs Berners-Lee terms “agents” to both apply metadata to documents automatically and to process documents with metadata already applied. The sources of these Semantic rules, Berners-Lee asserts, will be ontologies, which will in essence “[define] classes of objects and relations among them” (*Semantic Web 2001*). It is not clear whether Berners-Lee envisions these ontologies to be built by hand or automatically generated.

A follow-up to Berners-Lee’s original article, published five years later, gives more insight into the ways the Semantic Web has actually developed. Particular focus is given to the development of official “tagging languages” such as RDF and OWL. The article notes that the Semantic rules that these languages support have extended past Berners-Lee’s original “flat” conception of familial-like relations (sister, friend) or sub/super relations (owner, instance of): newer relations include “causal, temporal, and probabilistic logic” (*Semantic Revisited 2006*). Despite this extension, the basic triple of “entity, relation, entity” remains intact. Of particular note is this article’s discussion of ontology generation: while it still appears that ontology development is

considered to be a manual and not automated task, the article also recognizes the "often quoted concern about the Semantic Web – the cost of ontology development and maintenance", and in response to this proposes that "shallow ontologies" – consisting of simple relational structures as opposed to the more complex "deep ontologies" – may mitigate these concerns (*Semantic Web 2006*). While developing even these shallow ontologies is considered a task for a human designer and not a machine, by simplifying a larger problem into simple subproblems they are at least laying the framework for future automation.

While these articles describe the theory behind the Semantic Web, one must examine actual implementations of semantic computing to truly gain an understanding of it. We will do this in the following two subsections.

2.2. Semantic Web Implementations - Hand-Built Ontology

An excellent example of "classical" semantic computing can be found in the Redada system, developed for the Italian government to attempt to detect money-laundering schemes by analyzing news articles. By tagging certain information in these articles, the Redada system is able to build relations between individuals and events and detect patterns that may indicate a laundering operation (*Redada 2010*). To accomplish this tagging, Redada utilizes a relatively complex ontology focused on tagging individuals and events within a judicial framework. By referring to this ontology, the system is able to apply metadata to a document, accurately building an "understanding" of the events described. Furthermore, with reference to the same ontology, the system can then build patterns and extract information from these patterns. The ontology used for all these tasks is handbuilt and consists of 555 classes and 2000+ aliases, and is "intended to capture the essential conceptual entities and relationships in the knowledge structure about anti-money laundering due diligence on companies and individuals" (*Redada 2010*). Clearly, then, this is a highly complex ontology. At present this is a good example of many

semantic computing efforts – an ontology is handbuilt to a certain, often highly complex domain and then applied.

Another, perhaps more interesting implementation of semantic computing is the Semantic TagPrint system, developed as a joint effort between researchers at Siemens Corporation and Bogazici University in Turkey (*Semantic TagPrint 2010*). The system is described in a 2010 paper and contains an architecture which is in many ways similar to the architecture of the GCKEngine architecture: it first extracts potential terms to tag documents with, uses an ontology to assign a weight to each term, and finally uses a "concept weighting" module to decide which of these weighted terms should be applied to the document. The TagPrint system uses an ontology called UNIpedia in order to weight its terms, which in turn derives its ontological information from the WordNet lexical database[1]. Unlike Redada's ontology, UNIpedia was not developed specifically for a single task and is instead more of a general dictionary that is intended to be used for general-use tagging. Interestingly, while the prebuilit ontology WordNet is used as its primary source for semantic information, it also utilizes Google popularity metrics and Wikipedia article relations to affect a term's weight. The paper describes its use of Google to resolve word sense ambiguities: after twice querying WordNet, it retrieves and examines the top twenty results from Google for some query to effectively disambiguate the word (*Semantic TagPrint 2010*). While this use of Google to gain semantic information is similar to our own approach, the TagPrint system does not attempt to use Google to dynamically build ontologies, only to modify the results already retrieved from a handbuilt ontology. In other words, it is used to *disambiguate*, not to *discover*.

These are two examples taken from a much larger group of semantic projects using hand-built ontologies. As can be seen, much work has been done to power semantic computing using

[1] http://wordnet.princeton.edu/

handbuilt ontologies. It seems clear, then, that automatic ontology building can be combined with these automatic tagging / analytical techniques to produce truly powerful semantic computing solutions.

2.3. Semantic Web Implementations - Automatically-Built Ontology

While still a relatively newer field of study (or at least, less explored) than the automatic tagging discussed in section 2.2, the technique of automatically building ontologies has been the subject of a fair share of papers and projects.

There are many tools that attempt to partially automate ontology generation, such as Protégé or OnTex (*Knublauch 2003; Ganter* 2009). These tools do not fully generate an ontology on their own, but instead provide an engineer assistance with developing ontologies. They are comparable to modern programming IDEs such as Eclipse or Visual Studio – in these tools, while an engineer still must write the core code of an application, the IDEs are able to do much work in assisting the programmer by automatically generating supporting code based on the engineer-written code. Such as it is for Protégé, OnTex, and similar tools. OnTex, for example, works by directing the ontology engineer through a top-down process in which the program attempts to exhaustively elaborate upon general concepts provided by the engineer. The program essentially attempts to build a framework that the engineer can then go and fill out using heuristics or other methods. While programs such as these are not exactly full ontology-building algorithms in their own right, they do much to assist a user in creation of ontologies and so may be considered at least slightly part of the automatic ontology-building paradigm.

On the other hand, there are systems which attempt nearly full automatic ontology building – one such system, which partially served as an inspiration for our thesis – is described in the 2007 IEEE paper entitled “A Method of Semantic Dictionary Construction from On-line Encyclopedia Classifications”. As the title suggests, this system attempts to create ontological dictionaries by

referencing and deriving information from the Wikipedia online encyclopedia. The authors note that Wikipedia itself features a semi-semantic structure: one can determine, to some extent, the relation between two pages by examining the Wiki links between them, and one can examine the categorization of articles to learn about the relationships between the subjects those articles describe (*Encyclopedia Classifications 2007*). In some cases these relationships are made quite explicit – Wikipedia provides category pages with sub/superclass structures, which naturally lends itself to semantic analysis. Furthermore, Wikipedia even provides disambiguation pages that may be analyzed to learn about the different senses a term may be used in. Through the use of this information, along with analytical statistics and other methods, the authors of this 2007 paper are able to develop a system to build ontologies automatically to some extent. While the use of Wikipedia is innovative, it is also limited, as Wikipedia does not and cannot have pages for every conceivable concept at the level with which we require, especially for more advanced topics. For example, no online encyclopedia would feature a page for every Drupal[2] module or Java function, even though we may wish to build an ontology out of these modules and functions. Though Wikipedia has its benefits, we must move beyond the online encyclopedia for true automatic ontology building.

As stated, this thesis takes inspiration from many of these systems, especially the last one insofar as we use an external web resource (Google) to gain information about our terms and their relationships. While we believe that our use of Google to fully build a new ontology is a relatively novel approach, we acknowledge freely and appreciatively that this approach would not be possible without the previously accomplished work mentioned here, among others. Having acknowledged this, we may now move on to describing our system in depth.

[2] Drupal is a popular web content management system. http://drupal.org/

3. Implementation and Architecture

The architecture of this project can be roughly divided up into four parts:

1. Code that prepares the data for the ontology-building algorithm.
2. The ontology-building algorithm itself.
3. Code that parses the data returned by the ontology builder and tags the HTML files.
4. The Solr search engine that the tagged files are loaded into.

We wrote the code for parts 1 through 3. Our implementation was written in the Java programming language, using a handful of libraries that will be discussed below. Part 4, the Solr search engine, was written by the Apache Software Foundation[3].

The purpose of the first part of the architecture is to download a number of and to extract from the tutorials a large pool of terms that may or may not be related to the domain we are building an ontology for. Before we can extract the terms, we must classify the downloaded documents to ensure that only pages that have both the format and the content that we want are passed on to the term extractor. We do this by classifying the tutorials into three different categories: good, bad, or hubs. After classification, the pool of terms is created and stored, and the ontology building algorithm is called.

In part 2, the ontology building algorithm, we take each term from the pool and run various "Google Relationship tests" to build a weight for the term. By sending various queries to Google, we can analyze the results and determine whether the term is related to the domain. If so, subsequent tests can determine what kind of relationship the domain and the term have, and how strong that relationship is. This data is stored in various weights related to the term, and the term is output to a dictionary file to await further processing.

---

[3] http://lucene.apache.org/solr/

By the time we reach Part 3, every term in the pool of potential terms has been analyzed using ontology building algorithm and given a certain weight for each relationship category. We process the flat dictionary file and divide the terms up into the different relationship categories depending on their weights and other factors. This classification turns the flat dictionary file into a full ontology for our domain. Once we have done this, we call an HTML tagging function to classify the tutorials downloaded in Part 1 according to the ontology. This function produces as output an XML document formatted for use with Apache Solr.

In Part 4, we simply index the XML document into Solr, at which point we can see our results. Our Solr search engine has a relatively simple format: each tagged document has one entry in the Solr engine, which displays the name of the page, the URLs, and the tags for the document. The relationship tags are displayed as Solr facets, and so a user can utilize these tags to limit the results shown in Solr to only those results tagged with certain metadata.

Having described the high level architecture of the GCKEngine, we will now describe the functionality of each part in greater detail.

3.1. Main File

The entire program, encompassing parts 1 through 3, is orchestrated by the GCKEngine class, stored in GCKEngine.java. In this class, one may set the name of the system that one wants to download tutorials for (like “Drupal” or “java”) as well as an absolute link to the directory where all search engine files will be stored (henceforth referred to simply as the engine directory). These two parameters will be passed into every other part of the program when called, which makes it very easy to configure the engine to download, build and tag documents for a new system.

3.2. Web Crawler and Hub Extractor

The web crawler is found in the GCKEngine_Crawler class, stored in GCKEngine_Crawler.java. In the course of a full run of the engine, the crawler will be called twice: the first time, to download an initial set of documents from Google, and the second time, to download a second set of documents extracted from the “hub” documents of the first set.

The first time the crawler is called, we attempt to download the number of tutorials specified in a given number parameter. We do this simply by using an HttpURLConnection stream to search Google for the phrase “<system> tutorials”; from the search results page, we detect and download each link. Each link is stored it in a numbered file. The crawler contains logic to prevent downloading duplicate page, achieved by checking whether the URL is unique or not. In addition to storing each downloaded tutorial in its own numbered file, the crawler also produces a “downloaded resource file” text document to list the downloaded files. Each line of this text document contains the file number, page title, resource URL and filename of the downloaded resource.

After calling the crawler the first time, we classify the downloaded files into “good”, “bad” or “hub” categories. The classification system, as well as specific definitions of what these categories mean, will be described in section 3.2 below. After we have created a list of hubs, we attempt to extract the links in the hubs using the GCKEngine_HubExtractor class, stored in GCKEngine_HubExtractor.java. The hub extractor uses regular expressions to find every link in the document and identify both the URL of the link and the link text. Because there are certainly more links on a hub than just the list of content that we want to download, we need to determine whether each link is a good link or not. To determine this, we use the Boilerpipe[4] text

[4] http://code.google.com/p/boilerpipe/

scraper on each document in order to get the flat text of the main element in the document. For this and every other use of Boilerpipe in the engine, we use the CanolaExtractor scraper implementation, a SVM-trained extractor that was found to produce the most generous results while still cutting out most unrelated content. After retrieving the main content of each document, we check each link to determine whether the text of that link appears in the scraped main document. If it is found there, then we can say with relative confidence that the link is part of the hub, and therefore, is likely to be good. All the good links are added to an array, which is passed back to the crawler upon hub extraction completion. The crawler then simply downloads every link to its own file, and appends each resource's information to the aforementioned downloaded resources file. Once this has finished, the crawling stage is complete and we are ready to begin extracting information from the downloaded documents.

In practice, for the systems that we ran the engine on, we specified an initial download of 200 resources. For each system, after performing hub extraction and downloading the extracted links, we generally ended up with around 1,000 total downloaded documents. This was found to be an optimal number of resources, being both large enough to produce decent results, while small enough to be manageable for this thesis.

3.3. Classifier and Hand Classifier

After we have downloaded the documents, we must classify them. We only want to extract terms from documents that have a chance of containing good terms related to the system. A large part of the success of the automatic ontology builder depends on it having a large pool of potentially good terms to classify; it will not work as well if the documents used to create that pool of terms are not relevant to our system. So, the classification step is very important. Our engine classifies documents into three separate categories, which we have defined in the following way:

1. *Good*: documents which are in general related to the system, and contain some type of information that would help a person to complete a task. This may be a step-by-step, a video tutorial, or a question-answer page.
2. *Hub*: documents that are in essence a large, formatted list of links to other pieces of content. A hub document does not necessarily need to be related to our system or not, and so we do not care if a hub is "good" or "bad." That is to say, we care more about accurately classifying a page as a hub than we do about its content being good or bad, because after we extract the documents from the hubs (this process is described in the previous section) we can classify those extracted documents as good or bad anyways. Note that there are some documents we classify as *good* that may look like a *hub*: for example, a page with certain important content, followed by a list of links. If the page contains informative content, we want to classify it as good even if a list of links appears later in the page, or even if the content is arranged in a large list-like format. Only if the list of links does not contain any significant content by itself, but instead simply contains links or other insubstantial content, do we classify it as a *hub*.
3. *Bad*: bad documents are any documents that are neither *good* nor *hubs*.

Our ultimate goal with the classifier is to create three text documents to be referred to in future parts of the program, each of which contains the document numbers and information about the documents classified into three categories. The GCKEngine program contains two ways of doing this:

First, we can use the standard classifier, stored in a class called GCKEngine_Classifier, found in GCKEngine_Classifier.java. This program uses the Weka machine learning software[5] to attempt to classify the documents. In order to use Weka, we must have three pools of "training files" to

[5] http://www.cs.waikato.ac.nz/ml/weka/

read into the classifier, one for each category. After we have read in the training files, we then pass each of the downloaded tutorials into the Weka classifier – more specifically, a multinomial naïve Bayes classifier. This classifier gives each tutorial a weight for the three categories, and we classify the document into whichever category has the highest weight. The performance of this automated classifier is documented in section 4.2.

For the systems that we indexed into our engine, we did not actually use the automated classifier. The reason for this is that in order for the Weka classifier to accurately classify downloaded documents into the categories, it would have required a hand-classified base of trainer files of sufficient size. The number of handclassified tutorials Weka would require to properly train the classifier is approximately the same size as the number of tutorials we have downloaded in the first place – which was usually around 1,000, as noted in the previous section. Due to this fact, we did not use Weka to classify our tutorials, and instead simply handclassified them by placing them into one of three separate folders, one for each category. We created a program to create the required text files containing the lists of classified tutorials that GCKEngine_Classifier would have produced. This program may be found in the GCKEngine_Handclassifer class, in GCKEngine_Handclassifier.java. In addition, this program performs a second wave of duplicate checking, to eliminate any duplicates the Web Crawler may have missed.

Regardless of which classification method we use, in the end we will have produced three text documents containing the names and numbers of the files classified into the categories. After we perform hub extraction and download a second wave of tutorials (described in section 3.2) we classify these documents as well. We are then ready to move on to the ontology-building algorithm.

3.4. Term Extractor

The term extractor is run after the classifier, for the purpose of extracting potential terms from the good documents. At this stage, we do not attempt to determine whether the terms we are extracting are related to the domain or not; that will be performed in the ontology builder, as described in section 3.5. Here we are only concerned with amassing a pool of terms from the downloaded HTML files. In addition to the downloaded HTML files, we also use terms derived from the Google Adwords keyword tool[6], which provides “related keywords” for searched terms. The purpose of the Adwords keyword tool is to help people choose related terms to target in advertising, but it is also useful for this thesis to find terms that Google has determined to be related to our system. For every system we ran through our engine, we used this keyword tool to download terms related to the phrase “<system name> tutorials”, and passed these downloaded terms to our term extractor.

For both sources of potential terms, we want to eliminate duplicate terms, so that our ontology builder will not run the same term twice. We also want to standardize and clean up the terms by converting them to lowercase and removing all non-word characters (symbols, etc.) This is accomplished in two places.

For the terms extracted from the downloaded HTML files, we use the GCKEngine_HTMLParser class, located in GCKEngine_HTMLParser.java, to read in each document. We then use the Boilerpipe web scraper (described in section 3.2) to get the main content of each document, formatted as plain text. We extract the first 100 words in the document and add these to the list of potential terms. We only extract the first 100 words for two reasons: first, the principle of locality indicates that the important terms in each document will likely appear early on in the

[6] https://adwords.google.com/o/Targeting/Explorer?ideaRequestType=KEYWORD_IDEAS

body. Second, we cannot afford to search every single term in every document, as this would consume enormous amounts of time and resources. 100 terms was chosen as a compromise between getting as many important terms as we could, and keeping the number of terms the ontology builder needs to examine manageable. After we have extracted the terms from each HTML document, the terms are appended to a raw terms file, which will later be parsed when we run the ontology builder.

For the Adword keywords, term extraction is accomplished in the same class as the ontology builder, and takes place right before the builder runs. Because we are only parsing a single file, Adword term extraction does not need its own class. Additionally, we do not limit the number of terms extracted from the Adword keywords results: this is because the terms are more likely to be related to the domain (since they have already been matched to the system by Google's algorithm), and also because there are not very many terms in the Adword keywords files anyways. Because the Adword keywords terms are extracted in the same class as the ontology builder, we do not explicitly append them to the raw terms file; however, these terms may be added to the raw terms file if the ontology builder fails in the middle of classification.

Once we have extracted the terms, we are ready to proceed to the heart of the GCKEngine – the ontology builder.

3.5. Ontology Builder

The purpose of the ontology builder is to take the pool of raw terms built in the Term Extractor, and to give each term a weight for each of the possible relationships it could have with the domain. Despite its name, the ontology builder class comprises only one half of the ontology-building algorithm; the other half is the ontology classifier, which will take these weights and determine which of the possible relationships each term has with the system. In doing so, an ontology for the domain, albeit a very basic one, is effectively built.

The ontology builder is located in the GCKEngine_OntologyBuilder class, which may be found in the file GCKEngine_OntologyBuilder.java. This class contains a function, “buildRelations”, which takes parameters indicating the source of the terms we will attempt to weight (HTML files or Adword keywords), the name of the system, the engine root directory. This buildRelations function is what is called from the main file, described in section 3.1; however, the core function of both the GCKEngine_OntologyBuilder class and the ontology building algorithm in general is a second function called “googleRelationships”. It is this function that queries Google and returns an analysis of the results for each term. We will first explain how this googleRelationships function works, and after this we will explain how we use it in the buildRelations function.

3.5.1. The googleRelationships Function

The googleRelationships function is designed to be a very general function. Its purpose is to determine the strength of relationship between terms – it checks either two, or more, terms per call. It does this by querying Google with a search string made up of the relevant terms. The results of this query are then analyzed using a regular expression pattern, and two metrics are produced. These are:

1. Coverage: out of the results returned, the coverage denotes how many the pattern appears at least once in.
2. Frequency: out of the returned results, the frequency denotes how many times the pattern appears in total.

High coverage and frequency naturally indicate a strong degree of relationship between the two terms, while low coverage and frequency indicate the opposite, a low degree of relationship. We can use this general function in many different ways: to establish relationships between the domain and a term, as well as to check whether that relationship is of a certain type (i.e. a component, or a version.)

To perform the search, we use the Google Custom Search API[7], a paid service offered by Google, which allows us to perform a Google search and receive the results in a standard format (JSON[8]). We could, of course, use an HttpUrlConnection stream to download the HTML of the Google results, as we did in the Web Crawler (detailed in section 3.1); however, the ontology builder requires extremely high volume of searches (thousands of terms each requiring multiple queries), and Google blocks queries to its standard frontend at http://www.google.com after a certain number of searches within a finite span of time. Therefore, we use the Custom Search API to allow us to build an ontology in a reasonable amount of time, which is worth the small fee charged by Google ($5 USD for 1000 queries). The fact that calling the googleRelationships function is expensive, both figuratively and literally, means that we wish to call it as few times as possible. It is for this reason that we put so much effort into ensuring that the terms we run queries on with this function have as good a chance of being good terms as possible.

Each call to the googleRelationships function must pass in two parameters: an array of terms, and a Java Pattern object that will be used to analyze the results of the query. Each term we pass in will be used to construct the query, which will be of the format *<term 0> <term 1> <term 2> … <term n>* - that is, each term will be displayed in order with spaces in between. After we have built the query string, we pass it to the Custom Search API; ten Google results, formatted in JSON, are returned. The Custom Search API currently only allows ten results to be returned per query, and additional results require additional queries to obtain. However, ten results is sufficient for our purposes; since they are the first ten results, they are theoretically the most relevant results for the query, and ten highly relevant results should give us as good an indication as we need of the relationship between the terms.

---

[7] https://developers.google.com/custom-search/v1/overview

[8] http://www.json.org/

We use the returned JSON string to create a custom object, of the class GCKEngine_SearchResult. This class is defined in GCKEngine_SearchResult.java, and contains variables for the coverage, frequency, and an array of GCKEngine_ResultItems, another custom class meant to store the details for each of the query results. The GCKEngine_ResultItem class (found in GCKEngine_ResultItem.java) stores the title, URL, and snippet of each of the query results. After parsing the JSON and creating the array of 10 GCKEngine_ResultItems, we call the calculateAttributes function in the GCKEngine_SearchResult object to calculate the coverage and frequency for this particular query. This is done using the Java Pattern object that was passed as a parameter into the googleRelationships function. After finding these attributes, we return the GCKEngine_SearchResult object with the GCKEngine_ResultItems, coverage and frequency for this particular query.

3.5.2. The buildRelations Function

The ontology-building buildRelations function in GCKEngine_OntologyBuilder calls this googleRelationships function a number of different times in order to weight each term. The core of the buildRelations function is a loop, with one iteration per term, in which the googleRelationships function is called one or more times. While the googleRelationships function is very *general*, the buildRelations function is very *specific* – that is, specifically formatted to use googleRelationships to determine weights for the predefined relationships we are searching for, all within the context of a certain system within our chosen domain. We attempt to show with the buildRelations function that the general googleRelationships function can be applied to a specific use. The remainder of this section will discuss how we do this.

For every term, we first perform a simple query with googleRelationships to try to establish whether a term and the system are even slightly related – in essence, a "go / no go" test. We do this by performing a query consisting of the string "<system> <term> tutorial", and searching for

the term as our pattern. If this query has a coverage of at least five or greater, then we move on to more specific tests; if not, we add the term to a list of rejected terms and move on to the next term. While this may seem like a very easy test to pass, in practice it rejected a surprisingly large number of terms, in particular many "garbage terms" consisting of misspellings or gibberish. By rejecting these terms out of hand, we are able to reduce the number of queries we make by a large amount. Note that we do not perform the "go / no go" test on terms with numbers in them; while these terms may in some cases be likely to fail the test, they may also be version terms and hence we want to preserve them.

If the term passes the "go / no go" test, we proceed on to establish its component, version, and provider weight. To establish the component rank, we perform two more queries: a low level test and a high level test.

The low level coverage test sends only the term to the googleRelationships function, and uses the system name as its pattern. If searching the term alone is enough to return the system name, then we can say with very high confidence that this term is in some way related to the system. This test will allow us to weigh heavily terms that are very specific; for example, the term "JFrame" or "AWT" for the Java system.

Not all important components for a system will be specific terms. Some may be terms that are quite general; an example of this is the "Views" module[9] for the Drupal system. "Views" is obviously a highly generic term; it will not pass the low level test above. In order to properly weigh it, we also perform a high level coverage test. This test sends both the system name and the term to googleRelationships, and uses the word "tutorial" as the pattern. By specifying the system name in the query, we are able to bias the results for the term towards the system

[9] Views is a highly important Drupal module: http://drupal.org/project/views

context; if within this context we see the term "tutorial" quite often, then it is likely that the term, though generic, is a component of the system.

For the final term rank, we use the following formula:

*termRank = (lowLevelTest.Coverage * 3) + highLevelTest.Frequency + goNoGoTest.Frequency*

As noted above, we weigh the low level search coverage very high. We do not weigh the high level search coverage very much, as it is possible for an unrelated term to get a high score on this test as well (this is especially true of certain common classes of words, such as pronouns or words like 'the' or 'a'). In order to further help weigh the generic components, we include the frequency from the "go / no go" search. If the term is good, then it is expected to have a very high frequency from this search even if it is generic, even higher than the extremely common, generic words mentioned previously. This formula has shown to produce very good results, as will be shown in section 4.3.

We have described how the test to determine whether an item is a component is performed; we now describe how the test to determine whether an item is a version or provider is performed. Note that we only perform these version and provider tests if the term passed the "go/no go" test performed earlier on.

The version weight requires a more specific test, since it is a more specific relationship. We only perform a single googleRelationships test to establish the version rank: this test uses a query string consisting of the system name, the term, and the word "version." The pattern we use for the version test is very specific: we search for the system name (preceded by punctuation or a space) followed by the term, with a space in between. We then use the results of this test to determine the version rank, using the following formula:

*versionRank = (versionTest.Coverage * 3) + (versionTest.Frequency)*

Note that we make a few assumptions about the versions in this algorithm. We assume that the common way that the version is written is following the system name: for example, “Drupal 7”, or “PHP 5.3”. We also assume that the version term has a number somewhere in it; in fact, we do not even perform the version test if the term is not at least partially numeric. We will discuss the validity of these assumptions in section 5.

Finally, we last attempt to determine the provider weight. Due to the nature of the provider term, we do not actually perform a new googleRelationships test to determine the provider weight, but instead use the results from previous tests. We gather the URLs from each of sets of results from the component and version tests, and check to see how many times the term appears in the top-level domain of each URL. For each appearance, we increment the provider rank. If the term is indeed a provider, then it is reasonable to believe that the term will be present in the top level URL; as with the other two relationship category tests, we will discuss the effectiveness of this design section 5.

After we have completed all of our tests, we take the term name and its weights, and store them in an object of the class GCKEngine_Dictionary, defined in GCKEngine_Dictionary.java. We then check to see if the object has a weight greater than zero for any of the three categories; if it does, then we add it to our official list of accepted terms, but if it does not, we reject it and add it to a list of rejected terms.

This entire process is run for every term in the term pool. Once it is finished, we write out to “dictionary” text files the names and weights of all the accepted terms. We also write out the names of the rejected terms to a “rejects” text file.

We now have completed one half of the ontology-building algorithm. The second half, classifying the accepted terms into a full ontology, is performed next.

3.6. Ontology Classifier

The Ontology Classifier is used to examine the weights of each of the accepted terms and, based on these weights, to classify the terms into the different relationship categories. Like the buildRelations function in the Ontology Builder, the Ontology Classifier requires specific configurations per the domain. We must program our Ontology Classifier to be able to properly classify the terms into the right relationships: this is done by manually writing an algorithm to examine certain information about the term, and deciding on how to classify it. Again, this is a *specific* implementation and will have to be written for every domain the engine is run on; the logic for classifying terms may be very different between two different domains. The logic for classifying the terms may be as simple as placing the term in whichever category it has the highest weight in, or it could be more complex to take into account other factors. Our implementation falls into the latter category; its classification logic is more complex.

The Ontology Classifier itself is not a large program. It is stored in the GCKEngine_OntologyParser class, in GCKEngine_OntologyParser.java. Its core functionality consists of reading in the accepted terms file created at the end of the GCKEngine_OntologyBuilder function, and running each term through a loop to classify it. For our implementation, the terms may be classified as either components, versions, or providers. We do so using the following algorithm:

1. First, we check whether the term has either a length greater than two character, or has numbers in it. If it is not numeric but is also not longer than two characters, we do not classify the term. This check helps to eliminate terms such as single letters or other highly generic, useless phrases that were not rejected in the Ontology Builder.
2. If the term passes the test in #1, we first check whether its provider weight is greater than 8. If it is, than this means that the term was present in at least eight of the ten

result URLs when it was checked, which indicates highly that the term is indeed a provider. If this test is passed, we add the term to the providers list.

3. If the term does not have a high provider weight, we next check whether its component weight is greater than 2. If it is, then the term could either be a version or a component. We do a simple comparison on the component and version weights; the term is classified into whichever category it has a higher weight in.
4. If the term does not have a component weight greater than 2, it still may be a version. We check for this by checking whether its version weight is greater than 10. If it is, we classify it into the version category.

After each term has been run through this algorithm and classified, we are left with three lists of term, one for each relationship category. We write out the terms and their category weights into three separate “dictionary” files – one for components, one for versions, and one for providers. Upon completion of our Ontology Classifier function, our automatic ontology-building algorithm is completed, and the results are these dictionary files. We are now able to move on to using our automatically-built ontology to apply metadata to our downloaded tutorial documents.

3.7. HTML Tagger

Because we ultimately wish to index our data into a search engine, we need to apply metadata to the documents to help us more effectively search for them. For example, if a tutorial for a system is focused on a certain component for the system, or on a certain version of that system, we wish to append metadata indicating that focus. While we could take a naïve approach to tagging our HTML files with metadata from our ontology, and add a tag for each occurrence of any term in the ontology, this would almost certainly result in heavy “overtagging”. We also do not want to tag a document with a term that is used in a sense different than the one in which it

is used for our ontology to be tagged. So, an effective automatic tagging algorithm is required. To this end we have created the GCKEngine_HTMLTagger class, located in GCKEngine_HTMLTagger.java.

The basic functionality of the HTML Tagger is to find each ontology term that appears in the main body of a document, and to give it a weight based on its use in the document and the strength of its relationship to the domain (established in the ontology-building algorithm, as described in section 3.5). If the HTML Tagger functions well, then the terms which are important to the document will have a higher weight, while those that are not important or are used in a different sense will have a lower weight. Each document is scanned and tagged in the following way:

For each document, we read it in and get a scraped version from Boilerpipe. This gives us the body of the document in plain text. We sanitize the scraped body of symbols, both to make it cleaner for our tagging algorithm, and such that it can later be passed into the Solr search engine, which requires sanitized text data. Finally, we truncate the scraped body to 100 words, much as we did in the Term Extractor (see section 3.4.) The logic for truncating the body is the same as it was for the Term Extractor: the most important terms in a document are likely to appear very early on. The reduction in the number of terms the HTML Tagger must examine in a document also means that there are fewer generic terms that it may be mistagged with. After we have a sanitized, scraped, and truncated body, we perform the same sanitizing operations on the document title (we do not bother scraping or truncating it, naturally.) Once this is completed, we send the title and scraped body to be run through three loops: tagging both with terms from a relationship category in each iteration.

For the first category, components, we iterate through the entire components dictionary and see which component terms appear in the title body. We perform this matching algorithm in a

very specific way. First, we split both the title and the body up into arrays of single words, achieved by splitting on a space character. We then iterate through both arrays and perform a substring match between each word and the term. We perform substring matches to allow a term to match certain variants – for example, if the term is “view”, then performing a substring match will allow that term to match “views”. It is, of course, possible that our substring matching may inappropriately match a short term to a much larger word. To prevent this, we ensure that the matched term, if it is shorter than the word we are matching it to, does not have a difference in size greater than three. Performing this for both the title and the body gives us the number of times each term is found in each.

After we have found the number of appearances of the term in the title and body, it is time to determine whether this term is actually the focus of the document or not. We do this by checking three metrics. First, we check if tag appears at least twice in the body. Second, we check if the term appears at least once in the title. Third, we check to make sure that the term’s ontology-builder derived weight (which we will refer to as *relationship weight* to differentiate from the weight we are calculating in the HTML Tagger) is greater than provided *cutoff* parameter. Analysis of the automatically-built ontology has shown that while terms in the ontology with a high relationship weight are overwhelmingly good terms, as the weight decreases the proportion of good terms decreases, to a certain *cutoff* point at which there are more bad terms than good for each weight. We will discuss this analysis more in in section 4.3; for now, it is sufficient to say that at after that cutoff weight, the term is far more likely to be bad than good, and so we do not even bother attempting to tag documents with terms lower than that cutoff weight.

If a term appears either twice in the body or once in the title, and has a high enough weight, we go on to perform the following tests:

1. Tag location: we set the term location value to be higher the closer the term is to the beginning of the document, again utilizing the principle of locality. If the term appears as the last term in the document, it has location value 0. If it appears as the first term, it has a location value equal to the number of words in the scraped body.
2. Tag frequency: the number of times the term is found in the document. We find this for both title and body.

From these two metrics, along with the relationship weight of the term, we calculate the tag weight using the following formula:

*tagWeight = (bodyTagFrequency * relationshipWeight) + tagLocation + (100 if titleTagFrequency > 1);*

Note that we weight an appearance in the title quite high. If a certain term appears in the title, it is almost guaranteed to be important to the document. If it does not, we are still able to achieve a high weight by being close to the beginning of the body, or by appearing often.

For the second category, versions, we again loop through the list of version terms. For each version we find, we check to see if the title or body contains the system name followed by the version in question. This is consistent with the assumed version pattern that we used in the buildRelations function, described in section 3.5.2. For each time this pattern appears in the document, we add the version relationship weight of this particular version term to a running sum. After finding all matches in the document, we set our running sum as the weight for that version tag, and add the tag and the weight to another list.

For the third category, providers, we first get the base domain of the URL, minus the preceding "http" or "https". After we have done this, we loop through the list of provider terms and check to see whether each provider term appears in the base URL. If it does, we add the provider term and its associated relationship weight to a third list. Note that here, we do substring matching,

much as we did in the components test. This is so a provider name can match even if its URL contains more characters than just the provider term.

We are now ready to move on to choosing which terms out of the ones we weighted to tag the document with. Actual "tagging" of the documents is accomplished by creating an XML file formatted to be indexed into the Solr search engine; we map document information such as title, URL and body snippet to Solr fields to the XML, and term we are tagging the document with gets mapped to a Solr facet. More information about the Solr fields and facets will be covered in the next section.

For the component terms, we do not map every term we weighted in the HTML tagger to Solr facets. We only map those above a certain HTML tagger-derived weight. Our implementation found that 75 was a good cutoff weight; terms above this weight can confidently be said to be related to the document, while terms below this weight may not be strongly related enough to warrant a tag.

Unlike the components tag, our implementation does not perform a weight check on the versions and providers tags; instead, if the term was tagged earlier on in the HTML tagger function, we apply it to the document without more thought. This is because these version and provider terms have more specific requirements on being tagged in the document at all than the components; so, it is likely that if a version or provider is identified as a potential tag at all, that the tag is indeed applicable to the document. So, for the actual mapping of versions and providers to Solr fields, we simply map every term that made it through the HTML Tagger weighting process. Once we have finished mapping, the XML is appended to the Solr indexing document. This process is performed for every downloaded tutorial; once it is complete, we need only to index the document into our Solr engine.

3.8. Solr Engine

After we have created the Solr XML document with the HTML Tagger, we then load it into the Solr engine, called “indexing”. It then displays our documents and tags. A screenshot of the engine can be seen below.

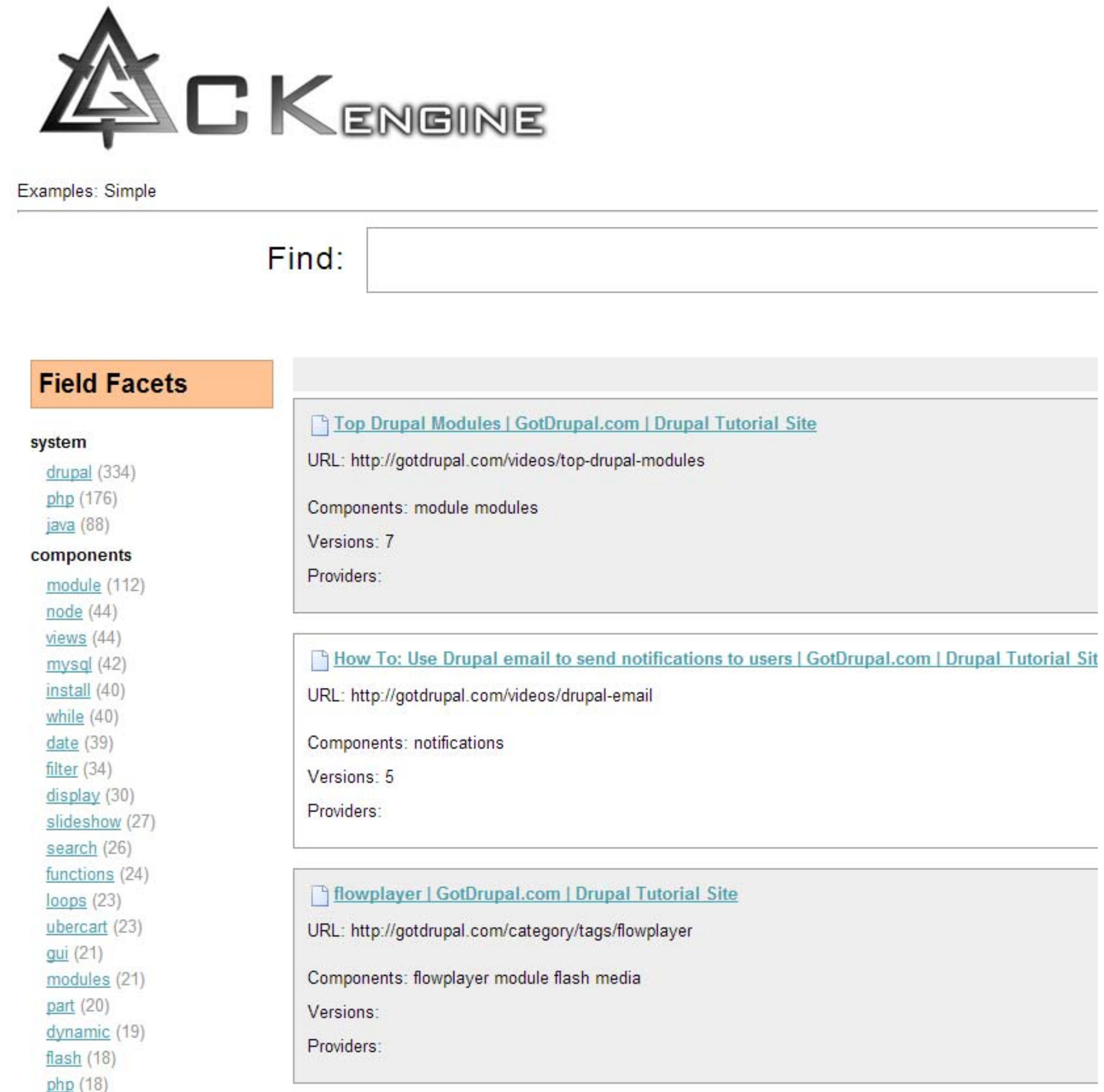


Figure 3.8.1: A screenshot of the Solr engine, with results in the center and facets visible in the left sidebar.

As can be seen, there are two ways in which our information is displayed in the Solr engine:

1. Each tutorial we have tagged has a single result that contains the page title, a link to the page, and a list of component, version and provider terms that the result is tagged with.
2. Solr displays the tagged relationship information in items known as “facets”. These are clickable links that correspond to properties about the documents indexed in Solr. One can click on a facet link and limit the displayed results to only those that have those properties. In our case, we use the facets to display the components, versions and providers: clicking on a facet link for a component can show only the tutorials tagged with that particular component. Additionally, we have a “system” facet that allows us to limit the results to only those documents that were downloaded and tagged for that system.

By displaying our tagged HTML documents in a Solr search engine, we are able to very clearly see the effectiveness of our ontology building algorithm. The way in which we are displaying our results – with the relationships represented as facets in a Solr search – is a realistic scenario for an actual use of an ontology. The primary purpose of semantic computing, as we described in the introduction, is to organize data and facilitate its retrieval. Our search engine attempts to do just that: one can narrow down a results set by system, by component, by version, and even by provider without ever having to do a keyword search. On the other hand, one can also do a keyword search (this functionality is provided by Solr) and narrow down the results from the set using the facets. Thus, we have in a very real way used our automatic ontology-building algorithm to create a basic, but perhaps effective, semantic search engine.

4. **Performance Evaluation**

The GCKEngine is a very complex algorithm, with many different stages; there are hence many ways in which it could be evaluated. From the possible ways we could evaluate the performance

of the engine, we have chosen three tests to run which we believe provide both good coverage of the code (every high-level step except the Web Crawler is covered) and touches on metrics that are highly important to the quality of the automatically-built ontology and the resultant search engine. These three tests are:

1. To evaluate how well we could automatically classify the tutorials, given a handclassified set of training files.
2. To evaluate the quality of the automatically-built ontology by examining the terms it added to the ontology.
3. To find the precision and recall of the GCKEngine.

In order to perform these tests, we first ran our engine on three systems, fully downloading, classifying, building an ontology automatically, tagging, and indexing into Solr each one. These systems are: Drupal (a web CMS), Java, and PHP. After we had done this for these three tutorials, we performed the three tests on each one. Our results can be seen below.

4.1. Tutorial Classification Evaluation

As discussed in section 3.3, for our test data we did not use an automatic Weka classifier, though we did write code to integrate this program into the GCKEngine. Instead, we handclassified all of our tutorials as either good, bad or hubs. However, now that we have a dataset of handclassified tutorials for each of the three systems, it would be useful to know whether we could automatically classify downloaded tutorials for that same system. It would also be useful to know whether we can use the handclassified tutorials for one system to automatically classify documents for another system. If either of these possibilities shows good classification performance, then we could potentially tag and index far more documents with far greater ease than the initial documents indexed and tagged as part of this thesis.

To this end, we tested both of the possibilities. For the possibility that we could use the handclassified documents of one system as trainer files to classify more documents of the same system, we performed a “90:10 test” – we divide the handclassified documents up into ten equal, randomized divisions, and perform ten classification tests, in each one classifying one of the ten divisions by using the other nine divisions each time as trainer files. We can then compare the classification produced by the automatic classifier to the actual classification done by hand, to find the accuracy of the classification. This will give us a good idea of the average performance of a classifier using handclassified trainer documents on the same system. For the possibility that we could use the handclassified documents of one system to classify automatically documents in another system, we simply load the handclassified documents of system 1 into the automatic classifier and attempt to classify the documents of system 2. Again, because we have already handclassified the documents of system 2, we can find the accuracy of the automatic classification simply by comparing it to the handclassification. We performed both of these tests on each of our systems. The results can be seen below:

90-10 (Same System) Test:

System:	**“Good” Accuracy**	**“Bad” Accuracy**	**“Hub” Accuracy**
Drupal	85.53%	62.07%	52.00%
Java	72.34%	78.33%	52.95%
PHP	72.26%	41.43%	63.56%

Table 4.1.1: Average accuracy for each category from the 90-10 test

The accuracies reported here refer to the percentage of pages automatically classified into a category that had been handclassified into the same category. They are generated as the average of the ten runs performed in the 90-10 test. As we can see, the performance of the automatic classifier using the handclassified tutorials as trainer files ranges from decent to mediocre. In general, the “good” category had relatively decent accuracy while the “Hub” category had accuracy that was rather poor, hovering around classifying only one in two pages

that actually were hubs into the hub category. Further, two systems had the highest accuracy in the "good" category while one had the highest accuracy in "bad."

The nature of the three categories may help explain why we see the results we did. It is not surprising that the "Good" category has, in general, decent accuracy, as documents in the good category show a uniformity of subject matter, as well as (to some degree) a similarity of structure, which would seem to make it easier for the classifier to classify. This is not the case for either the "Bad" or the "Hub" categories. The Hub category will, by definition, have very similar structure; but its topic matter could be (and, in fact, is) wildly inconsistent. Furthermore, the only thing "Bad" documents have in common is that they are not "Good" or "Hubs", which is obviously not something a classifier can classify on. Thus, it makes sense that we would see lower accuracies for "bad" and "hubs."

While these results are not bad, they also do not allow us to say with great confidence that we could effectively classify new documents for a system using old handclassified documents from the same system as trainers. Further performance testing may be needed, however, to draw a true conclusion on this matter; this is, however, outside of the scope of this thesis.

Different Systems Classification Test:

Trainer System:	**Classified System**	**"Good" Accuracy**	**"Bad" Accuracy**	**"Hub" Accuracy**
Drupal	Java	72.77%	1.03%	65.98%
Drupal	PHP	74.66%	1.37%	55.65%
Java	Drupal	0%	100%	0.93%
Java	PHP	3.71%	98.63%	9.57%
PHP	Drupal	18.02%	2.42%	96.29%
PHP	Java	60.21%	2.06%	90.98%

Table 4.1.2: Accuracies derived from the tests run using one system to classify another.

Clearly, the results here are worse than those taken from the 90-10 test. For each of the three systems, when used as trainers to try to classify the other two, a clear pattern resulted of having

at least one category with abysmally low accuracy (0% to 4), and another category with decent to high accuracy (70% to 100%). The fact that one category has very high accuracy should not be taken as a good thing – what this indicates is that that the classifier classifies documents into the high accuracy category at an artificially high rate, at the expense of the other categories. In other words, almost every document, regardless of what its classification should be, is incorrectly classified into that one high-accuracy category. This pattern can be most clearly seen in both tests using Java as the trainer files, as well as the test classifying Drupal tutorials with PHP trainers. In each case, one category was extremely high, while the other categories were very low, which means that almost all the documents were classified into the high category. Interestingly enough, the two tests run with Drupal as trainers show two categories with relatively decent accuracy, and only one category with very low accuracy. This likely means simply that instead of one category incorrectly accepting tutorials from the other two, we have two categories are incorrectly accepting tutorials from the third.

Clearly these results are unacceptable for automatic classification; we thus cannot use the handclassified files of one system to attempt to classify another system. On its face, this should not be surprising; a classifier's criteria goes deeper than simply the structure of the document, and classifies on *content* as well. Thus, they work truly only in one domain; and as we saw with the 90-10 test, in this case it does not work so well anyways due to the nature of how our categories are defined, which perhaps is not very well suited to automatic classification.

It should be noted that both the requirement to classify tutorials in the first place, and the classification categories that we classify into, are linked heavily to the method of document gathering and the subject of our tests in this implementation, which is computer tutorials. However, the classification is a separate step from the actual ontology building, and thus may not be necessary in some other implementations. For example, the automatic ontology-building

algorithm could be used to build ontologies for sets of documents which are by their nature all "good", such as medical documents; no classification would be required for this use. Additionally: we acknowledge the fact that in our tests we used only one classifier, the Weka classifier, and furthermore only one particular implementation, the NaiveBayesMultinomialUpdatable classifier in the Weka Java library. It is likely that using a different classifier would likely give different results to our tests, perhaps even radically different answers. However, the classification is not the focus of this thesis – the focus is the automatic ontology builder, and the classifier is only a means to an end. To improve and fully automate classification would be a topic for a different project.

4.2. Ontology Builder Evaluation

In order to evaluate the quality of our automatically-built ontology, we must examine each term added to the ontology and rate it as either good, bad, or misallocated. We can further expand the "bad" rating into two types – bad generic, and bad specific. The definitions for each rating are as follows:

Good:

A good term is a term that we would want to tag documents with, and it is placed into the correct category. While it is fairly simple to understand what constitutes a "good" term for a version (a term is either a version of a system or it isn't), it is slightly less clear what a "good" term is for providers and components. For providers, while a term may appear in URLs for tutorials, we only want provider terms that are both full identifiable provider names and refer to significant provider.

A "good" component term is even more complex. For it to be a good component, it clearly must be related to the system, and it should take, in general, the form of a subpart of the system – be that a module, function, implementation, technique, concept, etc. However, a "good"

component term should also be somewhat specific to a system. It should not be a term that is extremely common to multiple systems – for example, “String” or “Array” would not be “good” terms, as they are components of multiple systems. This is not an absolute rule, however – “MySQL” might be a good component term for both Drupal and PHP. A component should hence strike a balance between value and specificity.

Above all, it is important to keep in mind that for a term to be “good”, it must be something that a user would actually wish to filter tutorials by. If the term is not useful for this purpose, then it is not a good term.

Bad Specific:

A bad specific term is one that is not related to the system, and is not something that we would expect would be found in many documents and hence is not at risk of corrupting our tags. Gibberish words, usernames, misspellings, place names, and other specific items such as these are examples of bad specific terms.

Bad generic:

A bad generic term is one that is not related to the system, or is too generic to be included as a good term. Unlike bad specific terms, bad generic terms may be expected to be found in at least some of our documents, and hence are particularly at risk for appearing in our tags despite being bad. This is especially true of terms too generic to be included as good, such as “String”.

Misallocated:

A misallocated term is one that would be considered good, but has been assigned to the wrong category. While it is unlikely that a misallocated term would cause tag corruption, it also means that we will not be able to use that tag for its proper function.

While naturally we would prefer if all of the terms in the ontology were good terms, this is a virtual impossibility with an automatic ontology-building algorithm. It is for this reason that we give weight to the terms in our ontology: we aim our design to give only good terms high weight, while bad terms should “sink to the bottom” of the list with low weights. As mentioned in section 4.2, if we can achieve this, then we can find a “cutoff” weight above which terms are mostly good and below which terms are mostly bad, and use this cutoff weight to help keep our tags good.

We performed two tests on the ontologies for each of our systems: term ratings by bin, and term ratings by weight. The description of these two tests is as follows:

Bins:

We divide our terms into groups or “bins” of 100, where bin dispersion is ordered by weight. So, the terms in the first bin are the 100 highest weighted terms, the terms in the second bin are the second 100 highest weighted, etc. This will allow us to see the numeric distribution of good and bad amongst our terms; if our engine produces a good ontology, then there should be a sharp transition from mostly good terms in a bin to mostly bad terms, with only one bin at most in between being “mixed” We desire this pattern because it means that the good and bad terms are not mixed together – that we have a block of high-purity good terms, followed by a block of high-purity bad terms. If this is so, we can simply limit our accepted tags to those from the high-purity good terms block.

Weight:

We measure the percent of the total number of terms each category occupies for every weight level from maximum weight to minimum weight. That is, for a certain weight, we find the percentage of terms with that weight which are good, the percentage that are bad, etc., all adding up to 100%. Performing this test allows us to see how the terms are distributed by

weight. If our ontology is good, then we will see the high weights having high percentages of good terms and low percentages of bad. As the weight decreases, we would expect to see the good percentage decrease, until at some weight the bad percentage would overtake the good percentage. The weight at which this happens would be the aforementioned “cutoff” weight.

We performed these “bin” and “weight” tests on the three systems we ran through the engine. The results can be seen below. We first display and discuss Drupal results, then Java results, then PHP results. We will analyze these results in Section 5.

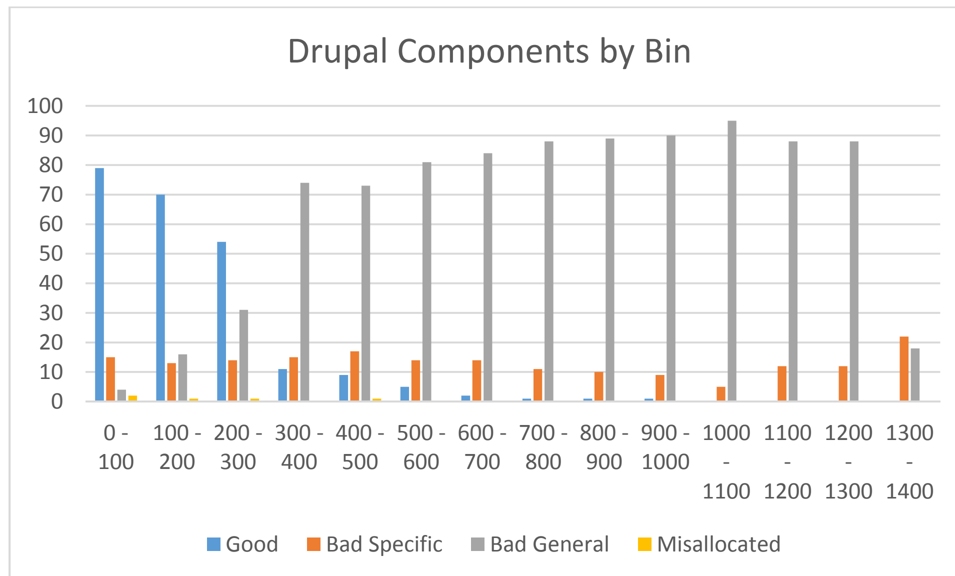


Figure 4.2.1: Drupal Component Terms evaluation sorted by Bin

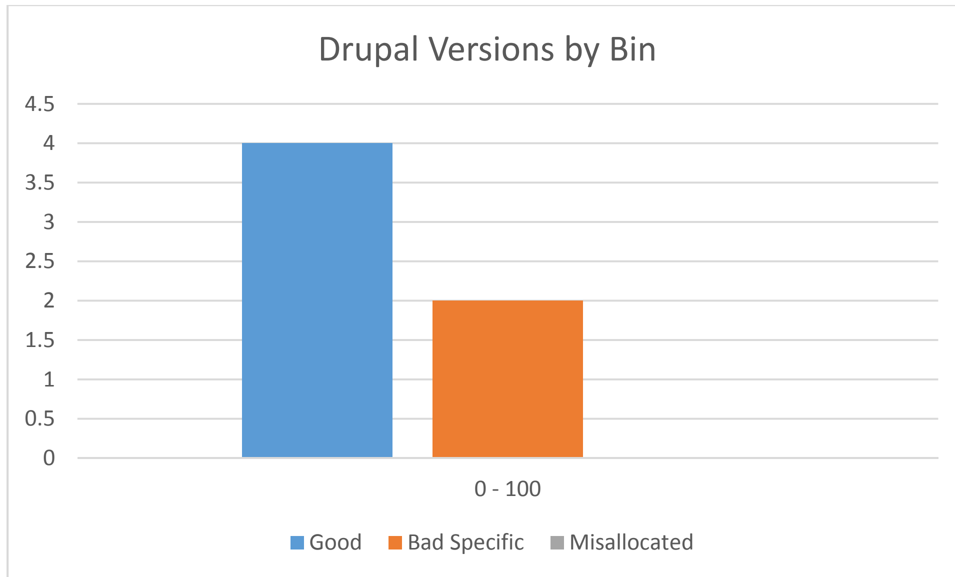


Figure 4.2.2: Drupal Version Terms evaluation sorted by Bin

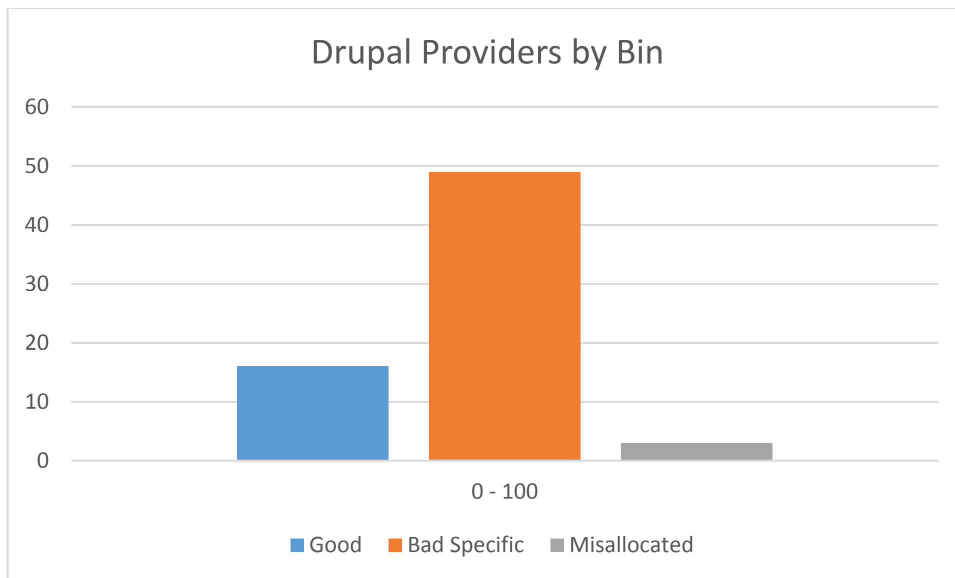


Figure 4.2.3: Drupal Provider Terms evaluation sorted by Bin

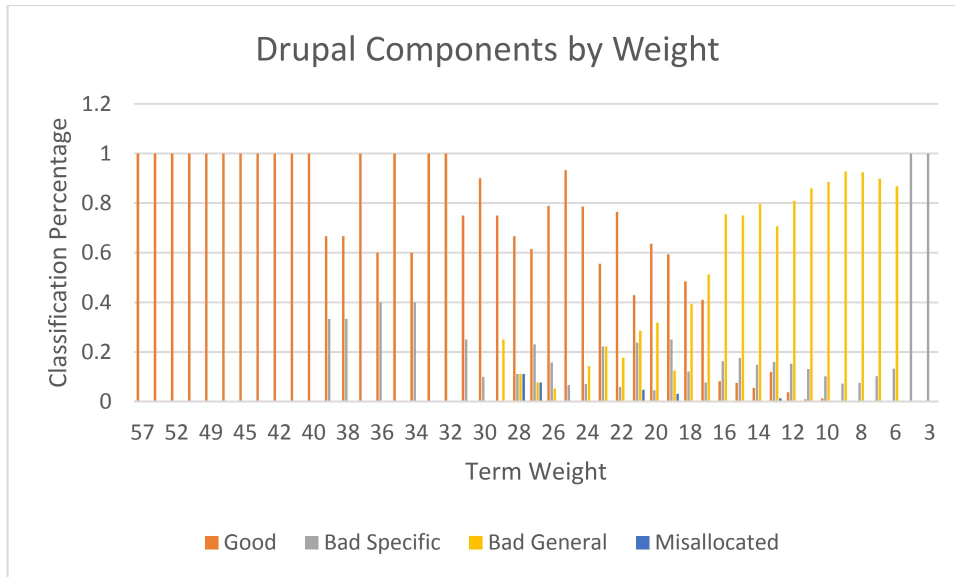


Figure 4.2.4: Drupal Component Terms evaluation sorted by Weight

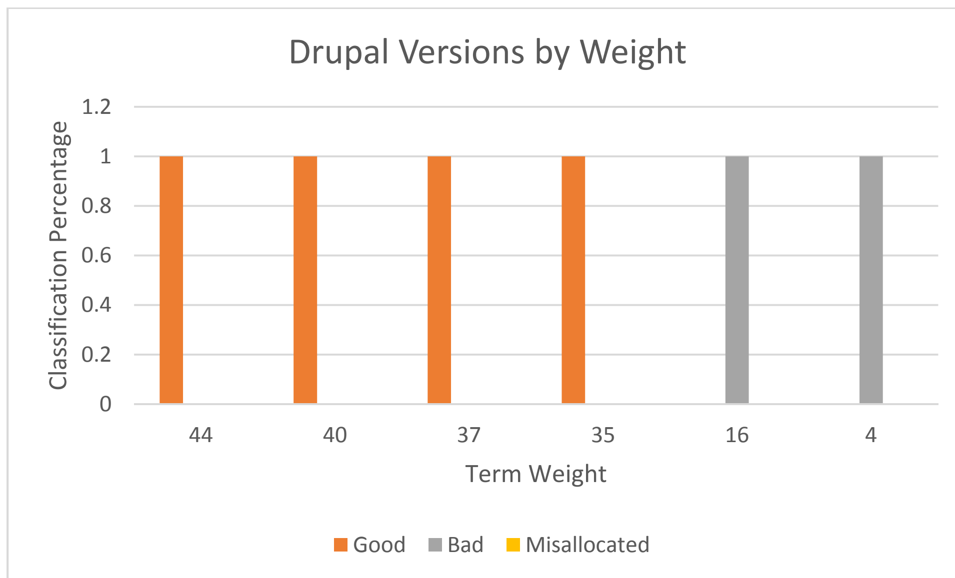


Figure 4.2.5: Drupal Version Terms evaluation sorted by Weight

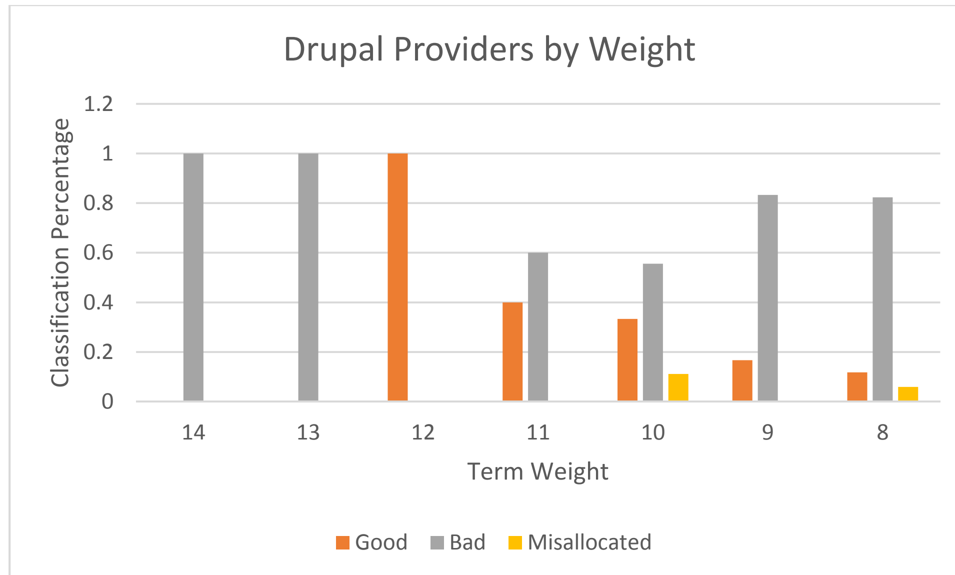


Figure 4.2.6: Drupal Provider Terms evaluation sorted by Weight

For the Drupal system, the components terms show exactly the kind of behavior we want to see. As shown by Figure 4.2.4, the high weights contain almost exclusively good terms. We start at weight 57 and it is not until after weight 32 that we begin to see any significant percentage of bad terms. From weight 25 to weight 17, the percentage of good terms steadily drops, and it disappears almost completely after weight 17. Before weight 17, there is no weight except good that reaches over 30%. By examining Figure 2.4.1 we can see that bins 0-100, 100-200 and 200-300 consist mostly of good terms, with most other ratings not reaching past 15 terms in the bin; the highest non-good rating in these three bins is “bad generic” in the third bin, at around 30 terms. In bin 300-400 we see a dramatic reversal, with bad generic dominating the bin and only about 10 terms in the bin considered good. Every bin after this is almost entirely bad generic, with only a small few terms in each bin being good (if there are any at all.)

The Drupal system also performed well in versions terms. Six versions total were found, four of which were rated good, as seen in Figure 4.2.2. Figure 4.2.5 shows that these four versions were the highest weighted, and all had much higher weights than either of the two bad versions.

Drupal Providers did not perform as well as the other two, but still performed acceptably. The weight spread of the providers, as seen in Figure 4.2.6, is a bit unusual: the first two weights are 100% bad. However, after this, we do see a similar pattern to the components terms, where we have heavy good, which transitions to heavy bad. While this is a desirable trend, the graph overall shows heavily mixed term ratings instead of distinct blocks, and for all but one weight the bad percentage is greater than the good. Providers, then, were of a far more ambiguous quality.

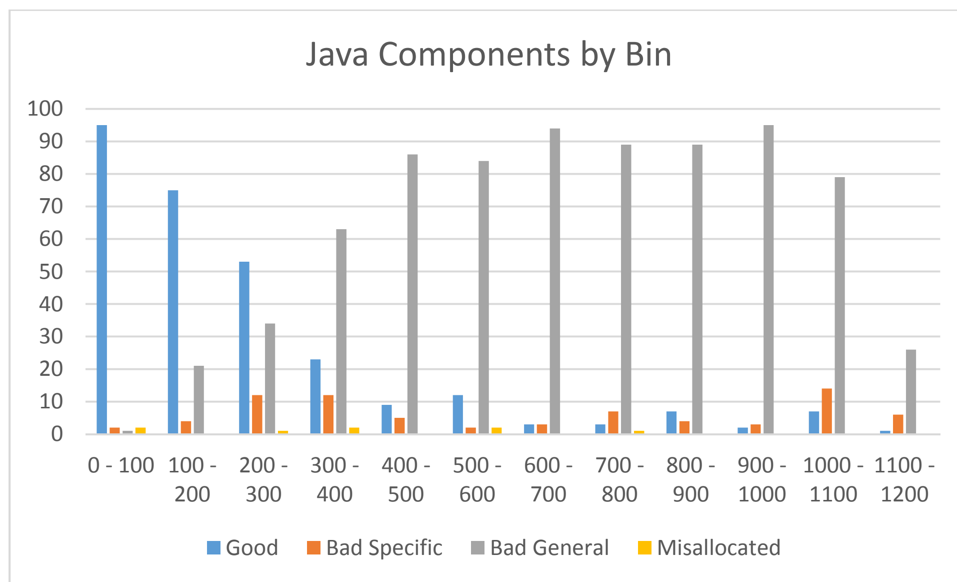


Figure 4.2.7: Java Component Terms evaluation sorted by Bin

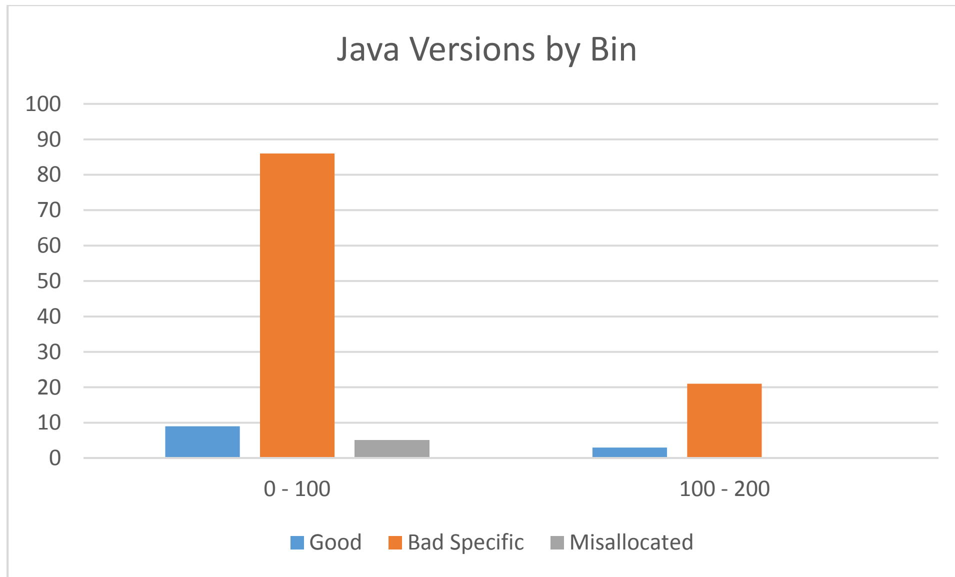


Figure 4.2.8: Java Version Terms evaluation sorted by Bin

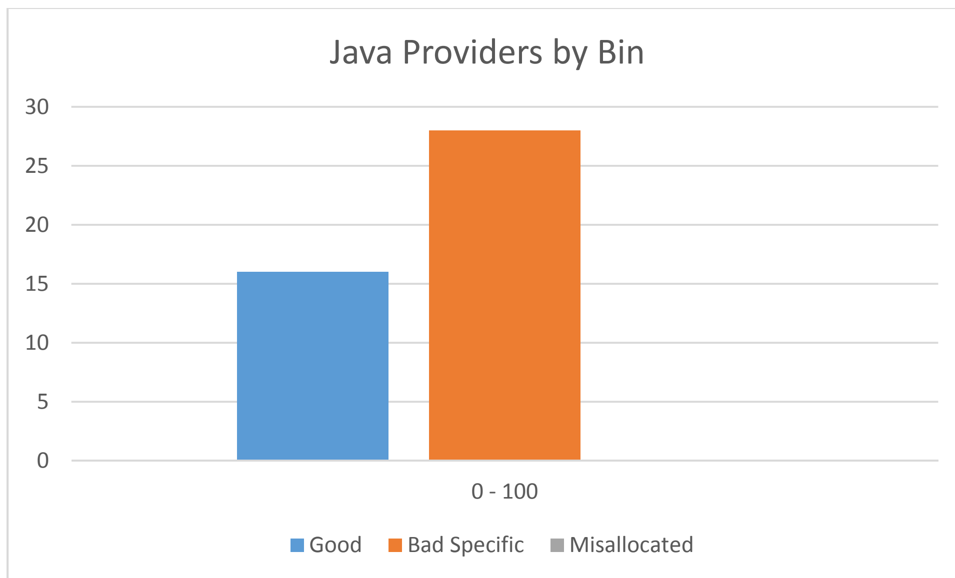


Figure 4.2.9: Java Provider Terms evaluation sorted by Bin

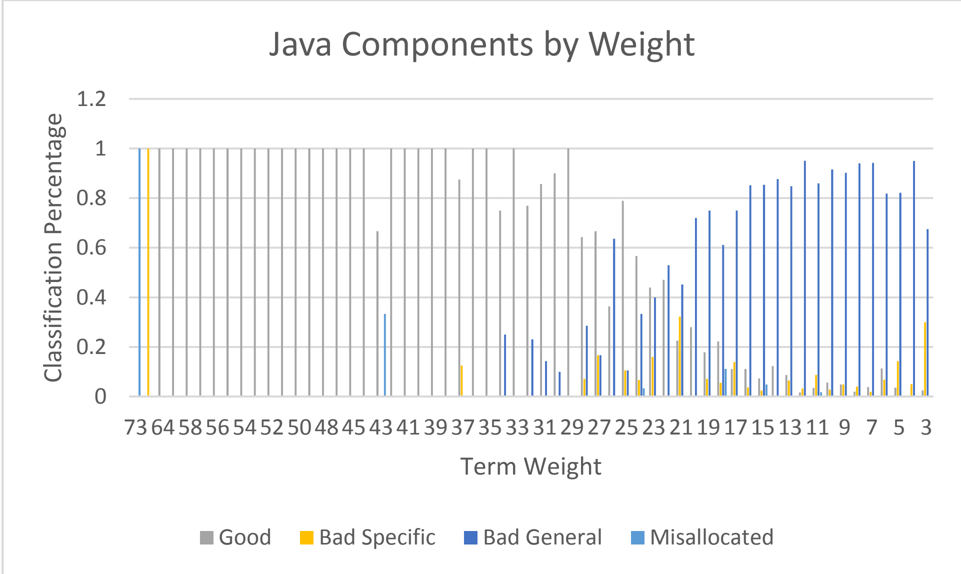


Figure 4.2.10: Java Component Terms evaluation sorted by Weight

Figure 4.2.11: Java Version Terms evaluation sorted by Weight

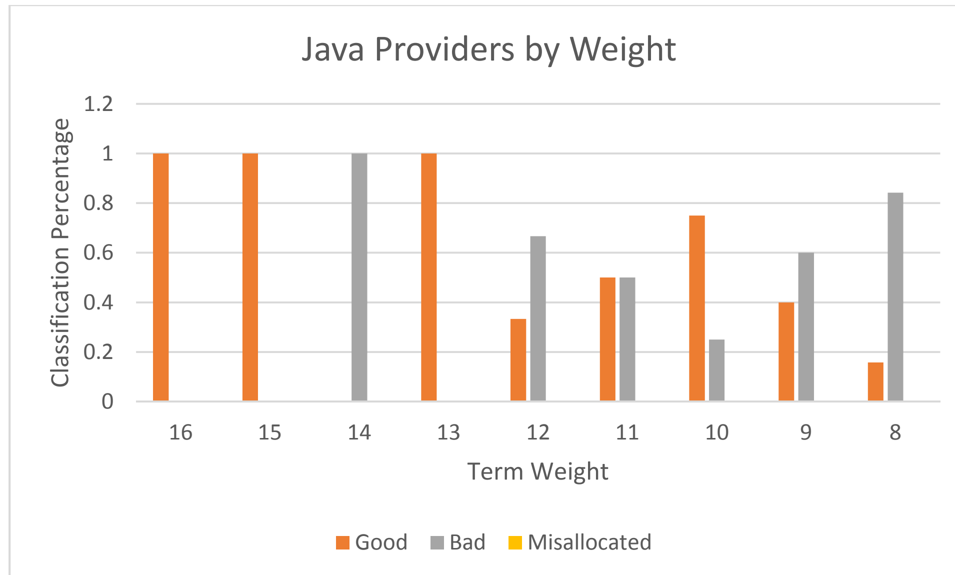


Figure 4.2.12: Java Provider Terms evaluation sorted by Weight

Once again, for the Java component terms, we see excellent performance. As before, Figure 4.2.10 shows the high weights are overwhelmingly good, and when sorted by bin we see a very clear divide between good terms and bad, as shown in Figure 4.2.7. Components, then, continues to show very good results. Unlike the Drupal terms, however, we see shockingly poor results for the Java versions. As Figure 4.2.8 shows, the vast majority of version terms are rated as bad for Java; out of approximately 130 version terms, Java has barely more than ten good terms. It is notable that when sorted by weight, as seen in Figure 4.2.11, we see that the early high weights are 100% good, and the rest of the weights are almost uniformly 100% bad. Despite the seemingly acceptable performance by weight, the terribly low number of good terms is cause for concern. We will specifically analyze this in Section 5.

Performance of Java providers was more acceptable, however. As Figure 4.2.9 shows, there were about half as many good provider terms as there were bad; Figure 4.2.12 shows that the good terms in general had higher weights, while the bad terms in general had lower weights. This is the pattern that we desire.

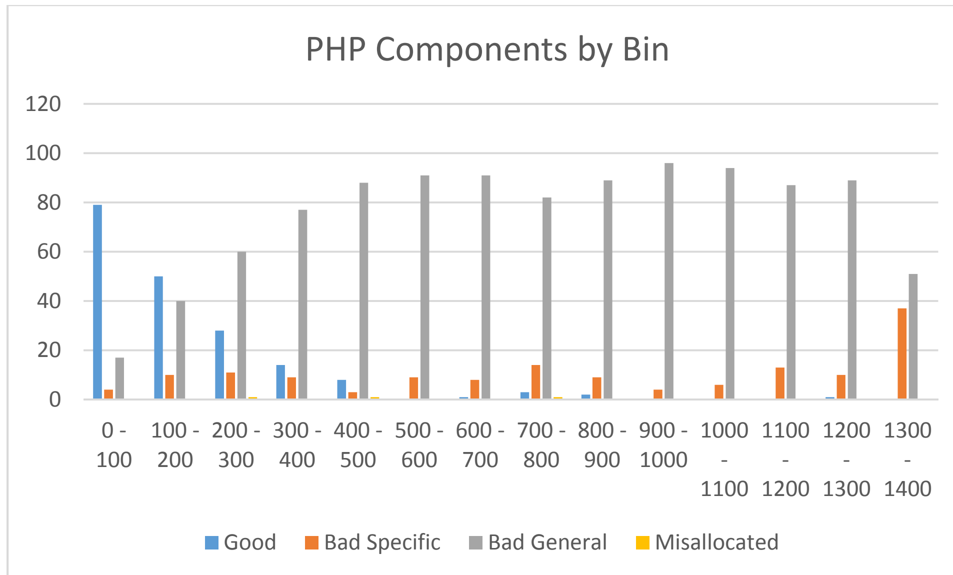


Figure 4.2.13: PHP Component Terms evaluation sorted by Bin

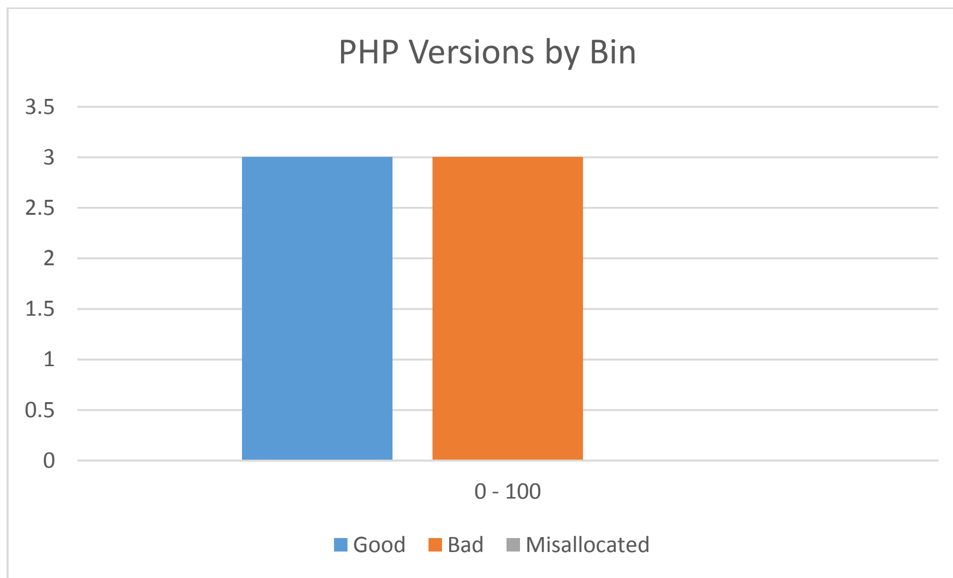


Figure 4.2.14: PHP Version Terms evaluation sorted by Bin

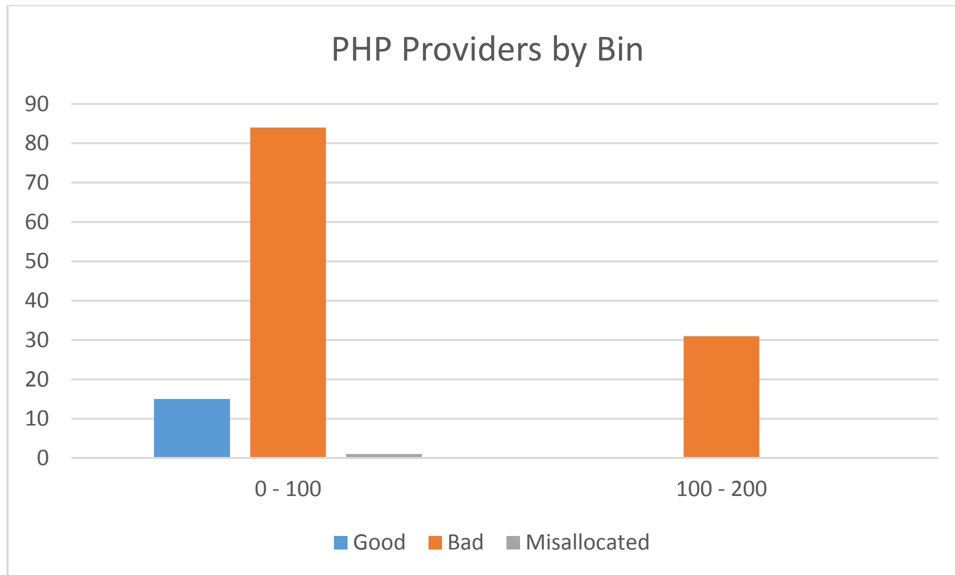


Figure 4.2.15: PHP Provider Terms evaluation sorted by Bin

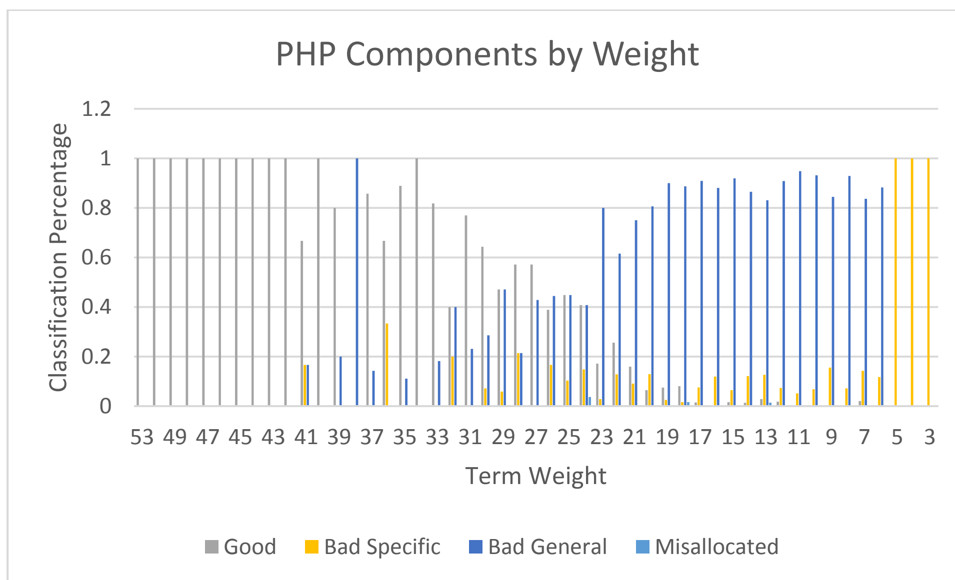


Figure 4.2.16: PHP Component Terms evaluation sorted by Weight

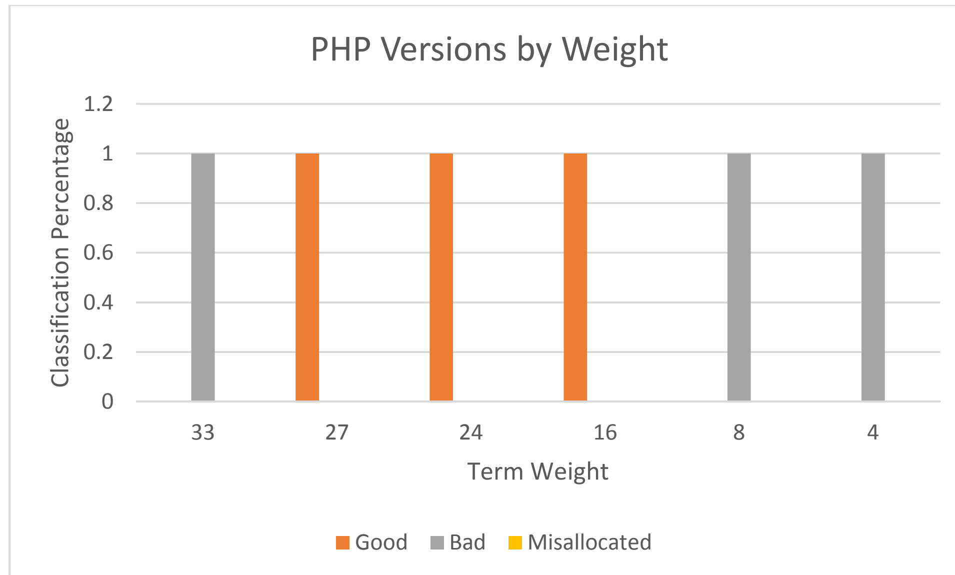


Figure 4.2.17: PHP Version Terms evaluation sorted by Weight

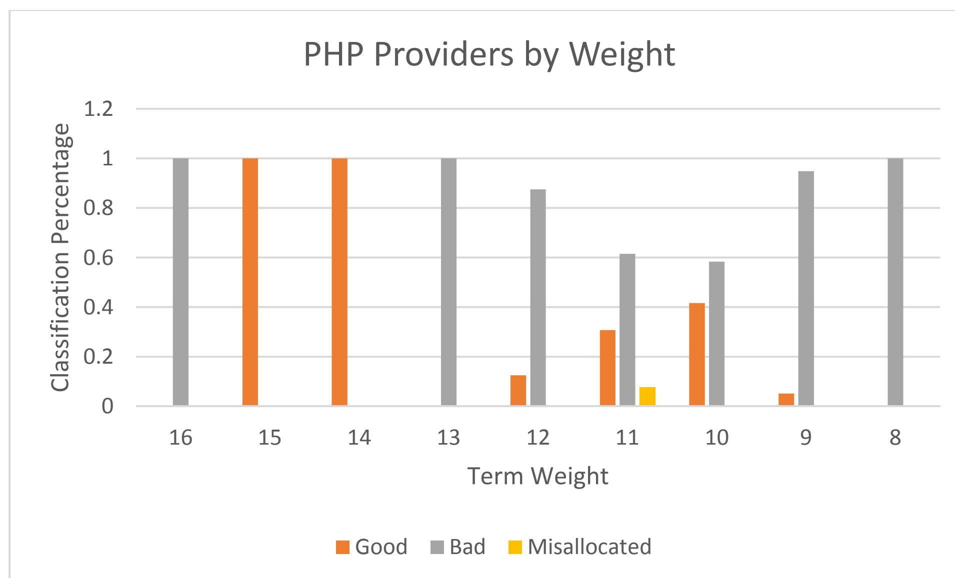


Figure 4.2.18: PHP Provider Terms evaluation sorted by Weight

As with the other two systems, we can see in Figures 4.2.13 and 4.2.16 that the PHP Component terms are being weighted as we would like them – a block of high-weighted terms at the beginning. Interestingly, the break between good and bad, while solid, is not as sharp as it was with either Drupal or Java. Also, there appears to be fewer good terms for PHP than in Drupal or

Java, and the good terms do not take up as large a percentage as they did in these two systems. However, this simply means that the cutoff weight must be set slightly higher when running the HTML Tagger for PHP.

For the PHP versions, we can see by Figure 4.2.14 that we have exactly the same number of good and bad version terms. Figure 4.2.17 shows that aside from an anomalous bad high-weight term, we have a block of high weighted good terms, followed by a block of lower-weighted bad terms, as we would desire.

Finally, Figures 4.2.15 and 4.2.18 show that we have a very similar situation with Providers as we did with Drupal. There are very few good provider terms for PHP; and while the few that are good are generally shifted towards high weights, the pattern shown in Figure 4.2.18 is definitely mixed.

4.3. Precision and Recall of the GCKEngine Search Engine

The final test we performed was a precision/recall test on the automatically-tagged tutorials. The precision and recall test attempts to rate the effectiveness of our overall tagging system, by analyzing, for each result, what percentage of the automatically applied terms are actually relevant (the *precision*) and what percentage of the terms which should have been applied actually were (the *recall*).

To perform this test, we tagged by hand each tutorial, providing tags in all three categories (component, version and provider); we then evaluated how our handtagged terms matched up with the automatically-applied terms provided by the HTML Tagger with the following formulas:

Precision = true positives / true positives + false positives

Recall = true positives / true positives + false negatives

Where:

True Positive: term which was automatically tagged and relevant.

False Positive: term which was automatically tagged but not relevant.

False negative: term that was not automatically tagged but was relevant

After getting precision and recall values for all three categories in each indexed resource, we then averaged these values together to get average precision and recall for the systems in our engine.

It is important to note that the terms we handtagged the documents with were not necessarily present in the automatically-built ontology. Thus, these tests are not just an examination of the HTML tagger but also of the automatically-built ontology. It is true that, if we wished to test the HTML tagger algorithm alone, that we should have limited ourselves to handtagging only those terms in the ontology; however, we are more interested in evaluating the GCKEngine as a whole with these tests, and not just the HTML tagger portion. The ultimate goal of this thesis is not simply to produce a good HTML tagger, or even simply a good automatic ontology builder, but to produce a quality automatically-ontology building algorithm that may be effectively applied to a real-world situation. In a practical situation, it will not matter if a relevant tag is missing because it was not in the ontology or because the tagger did not apply it; either way, it is missing. So, we attempted to test as rigorously as possible by applying the tags which we felt were most relevant, regardless of whether they were in the ontology or not.

Additionally: when calculating the precision and recall results, we occasionally performed the following changes to the data, when necessary. These changes were made to provide cleaner results.

1. If a document was automatically tagged with two variants of the same term – e.g. “module” and “modules”, or “inherit” and “inheritance”, we counted these two terms as a single term. We can justify this with the fact that automatic detection and

elimination these very similar terms from the ontology would be a trivial addition to the engine.

2. If the same document (determined by the URL parameter) appeared multiple times in the results, we only counted it as a single document.

The precision and recall test results are as follows:

System	Component Precision	Component Recall	Version Precision	Version Recall	Provider Precision	Provider Recall
Drupal	52.53%	73.21%	77.85%	89.25%	18.89%	21.17%
Java	55.17%	60.32%	46.59%	96.59%	7.95%	7.95%
PHP	39.68%	72.25%	88.82%	98.24%	36.47%	38.24%

Table 4.3.1: Precision and Recall of the GCKEngine

For the Drupal results, we can see that for components and versions our numbers are relatively good. The numbers are not so good, however, for providers. In all three cases, the recall percentage was higher than the precision percentage: this indicates that for this system, our engine applies more tags than are relevant. For components, our precision indicates that only approximately one out of every two tags is good; versions show better results for precision, with only approximately one out of every five tags being irrelevant. For providers, on the other hand, only one out of every five provider tags is *relevant*. As for our recall results: our component and version recall percentages indicate that for these categories, the engine is mostly able to apply the relevant terms to each document, failing to do so only one out of four times for components, and one out of ten times for versions. For providers, our recall numbers indicate that our engine is able to apply the correct provider only one time for every five documents. The data shows roughly the same trends for the Java and PHP systems. Java has roughly the same precision for components, but a somewhat lower recall; PHP, on the other hand, has a slightly lower precision, but roughly the same recall. Both Java and PHP have very high version recall, approaching nearly 100%. Java has relatively mediocre version precision, while PHP's

version precision is fairly decent. Finally, both Java and PHP, much like Drupal, have very low provider recall and precision, with Java showing the worst provider results of the three and PHP featuring the best at nearly 40% for both precision and recall.

Having heavily tested the GCKEngine, we may now analyze these results.

5. **Analysis**

Having performed the various tests on the GCKEngine, we may now analyze its overall performance and draw conclusions about both where it achieves its goals, and where it falls short.

As we can see from the tests on the classifier, it would be very difficult to automatically classify tutorials, even if we have a large base of handclassified trainer files. This is a disappointing result; an early goal for the engine was to have it be fully autonomous from start to finish, such that we could build an ontology and tag a set of classified tutorials simply by entering the name of the system we want to do this for. However, this disappointment is tempered by the fact that the results of the classifier are here largely a result of the systems we ran it on. It may simply be that tutorials for various computer systems are particularly hard to classify accurately; as mentioned in section 4.1, other domains which may utilize a GCKEngine-like automatic-ontology building algorithm may be easier to classify, or indeed may not even *need* a classifier.

From the results of our tests run on the results of the Ontology Builder, described in section 4.2, we can draw a handful of conclusions. The first and most obvious is that the Ontology Builder *is* quite good at determining a basic component-type relationship. Of the three systems we indexed, each one had component weight distributions showing a block of good terms encompassing the high weights, which quickly changes to a block of bad terms encompassing the lower weights. This means that, as designed, the good terms were almost universally more

heavily weighted than the bad terms. In this sense, the automatic ontology-builder was successful.

The version and provider term results, however, are not as clear cut. For versions, we have two apparent successes (Drupal and PHP) and one definite failure (Java). Why is it that the algorithm was able to successfully find and weight the versions for two systems, but was not able to find and weight versions for a third? We believe this is due to the assumptions we made about how a version would be formatted. Recall, as we described in section 3.5.2, that we assumed that a version would both contain at least one number in it, and would generally appear directly after the system name. These assumptions tend to hold true for both Drupal and PHP. The most recent four Drupal versions are simply titled and consistently referred to as Drupal 5, Drupal 6, Drupal 7 and Drupal 8. Similarly, PHP versions are generally referred to as PHP 4, or PHP 5. Because our assumptions held true for these systems, we were able to successfully classify and weight versions for them. Java, however, does not have such a clean and consistent pattern of version naming. There are two ways that Java breaks with our assumptions on version naming. First, many of Java's important versions use an acronym for the name preceding the version; J2SE 1.2 is an example of this. Because the word "Java" does not precede the version number 1.2, our version detector in the automatic ontology builder will not pick it up. On the other hand, some versions of Java do use the word "Java" for the system name, but the version number is not what we assumed it would be. For example, consider the version Java SE 6. While it has "Java" preceding the version number, the version itself is made up of two words, one that is numeric and one that is not. Again, our version detector is not designed to pick this up. It is not surprising, then, that our version functionality would fail to work if the assumptions behind it are incorrect.

The version results show the importance of accuracy in the relationship weighting programming we must do for the automatic ontology-building algorithm. For each domain we are building an ontology for, we must create functions which utilize the googleRelationships algorithm in a method effective to detecting the relationships we want to detect. We have seen that, if set up properly, the automatic ontology-building algorithm can be very effective. It is simply a matter, then, of writing effective functions for it.

This same principle can be seen when one examines the provider results. While none of our provider results were exceptionally bad, neither were they very good. Some provider terms represented actual providers, and some were simply generic terms that weren't providers at all. What happened? Again, like the situation with versions, this is the result of a relationship-detecting function that is not trained to full effectiveness. In this case, a specific problem may be recognized: we allowed terms to be considered a provider if they matched any portion of the base URL, instead of the entire base URL. This explains why many short, generic terms were classified as providers for all three systems. A more effective solution may be to require the terms to be longer to be considered a provider, or to match a certain percentage of the URL. A small change like this would likely improve the effectiveness of our provider-matching algorithm a great amount.

Finally, the results of our precision and recall tests may give us, to some extent, an evaluation of the GCKEngine overall. This evaluation is ultimately positive. These tests showed that our engine was, in general, very good at applying the majority of relevant components to a document, and even better at applying the proper version term.

For the components in general, we had a recall of roughly three terms out of four, which is indeed very decent. During the handtagging of the documents, we made the notable observation that in almost every case that a relevant term was not applied, this was because the

term did not exist in the ontology; this was true of all three systems. Furthermore, these missing terms were in general relatively few – for example, for Drupal, we estimate that if only five missing terms could have been added to the ontology, that the recall would have jumped to 80 or even 90%.

As we noted earlier, the precision for components is not as good as the recall. This is true for every category, but it manifests itself especially clearly in the components category. The engine habitually "overtags" a document. There are a number of likely reasons for this. The first can be seen from the observation that many of the irrelevant tags are relatively generic. As we saw in the analysis of the ontology builder, for each system we had at the high end of the weights a relatively pure block of good terms, and it is the low-weight edge of this block that we used for the cutoff weight in the tagger. "Relatively pure", unfortunately, does not mean entirely pure, and mixed in with these good terms were the occasional bad term. There were, in general, three "types" of "bad" terms that were applied to the document:

1. Highly generic terms: many of these wrongfully-applied terms were very generic, useless terms, such as "site". Often these tags had relatively high weights due to their generality.
2. Semi-generic terms: when we defined our categories for terms, we decided that we did not want to include terms which were not specific to a certain system but still applied – for example, terms such as "variable", "string", or "function". We decided that these terms were too generic to exist as tags, and so we considered them "bad" even in the cases where they *did* apply to a document (such as a tutorial on strings). It is likely that if we had allowed these terms to be considered as "good", that we would have higher precision and recall for all three systems.

3. Wrongfully-applied specific terms: sometimes a term that may have been a “good” term in other documents was wrongfully applied. This mistagging was generally due to the bad term’s existence in the document in a footer or sidebar that was erroneously picked up by the web scraper. For example, often in the footer of one tutorial were links to other tutorials on separate topics: occasionally these separate topic terms were picked up and applied to the current tutorial. This was especially prevalent in the PHP tutorials, as the majority of these tutorial documents came from a single site, which had a sidebar filled with links to other tutorial resources.

None of these problems are “core” problems. The high-weighted bad generic terms could likely be eliminated with tweaks to the ontology-building algorithm, and the improperly applied specific terms may be eliminated through improvement of the web scraper software. These improvements would not eliminate all sources of extraneous tags, but it would eliminate a large amount of them.

The version recall and precision is the highest of the three categories, for all three systems – the tagging algorithm was in general very good at detecting and applying the proper version when it was present, and not applying any version when no version was to be found. This success is likely for two reasons: first, unlike components, the requirements for successful version detection is much more specific. For a component term to be mistagged, that component word must simply appear in the document; for a version term to be mistagged, it must not only appear in the document but must immediately follow an instance of the system name. The second reason for the extremely high version accuracy is because for the Java and PHP documents, only very rarely were versions to be found in the documents at all! Our engine almost never mistagged a version when there was none, which naturally inflated the numbers for precision and recall. When a version tag *was* mistagged, this was generally because it

appeared (in the proper format) in a sidebar or comment that was incorrectly picked up by the web scraper. Even though this did happen occasionally, the high percentages for both version precision and version recalls indicate that it was a rare event in general.

We must account for the poor performance of the providers: this performance can be easily explained by the fact that the automatically-built ontologies for providers were poor in two ways. First, many of the terms rated as providers were in fact generic; second, many (even *most*) of the actual provider terms did not appear in the ontology. This led to a relatively high instance of mistagging (due to the generic terms) as well as a high rate of failure to tag (due to missing providers.) The high percentage of generic terms classified as providers is mostly the result of a poor provider detection algorithm in the automatic ontology-builder, but the more salient problem of lacking actual provider terms is due to the fact that very few providers were sent to the automatic ontology-builder for weighting in the first place. This is not surprising – for a term to be good as a provider term, it must be able to be detectable in a URL. However, experience shows that it is not common for the site name contained in a site's body (from which we gathered our terms) to be fully related to its URL, nor is it common for a site's URL to be present in its own body. These two factors add up to us simply not being able to find these provider names in the first place. If the provider names were passed to the ontology builder, it is highly likely that it would correctly classify them as providers, and our HTML tagger would likely tag them as such. Overall, while the poor precision and recall of the provider category is certainly disappointing, it is also very understandable after analysis.

The GCKEngine, in general, has higher rates of recall than precision. For our particular implementation, this is certainly preferable to the alternative: it is far better that we are tagging documents with the right terms, as well as a few extras, than to have low rates of extraneous terms (high precision) but also poor rates of correct tagging (low recall.) As we have discussed,

the extraneous tagging phenomenon may likely be solved with a little additional work; we certainly will not need to alter our core algorithms in order to do this.

Indeed, as these results show, our core algorithms are in general strong and good at the tasks they are intended to accomplish. As our ontology builder tests shows, the automatic ontology-building algorithm is able to build highly effective ontologies with the majority of good terms weighted highly. Furthermore, our precision and recall tests show that there are relatively few terms missing from our automatically-built ontologies, and that we are able to, in general, effectively tag documents with the proper terms. Thus, it may be said that we have been able to mostly achieve our stated goals with the GCKEngine.

6. Future Work

While the GCKEngine has achieved at least a relatively decent degree of success, it must be noted that it is also very limited in its abilities. The main goal of this thesis was to build an automatic ontology-building algorithm; and while we have done so, the current implementation limits us to a very specific type of ontology that can be built. In particular, the following limitations currently apply:

- Each term in the ontology may only be a single word, with no symbols.
- The created ontology is essentially a single level tree: there are only relationships between each term and the domain, and not relationships between terms.
- The ontology builder cannot perform basic term-variant matching or other simple language-relational functions – this is why our ontologies end up containing both the terms “theme” and “theming” with no relations between them.

It should be noted that there is nothing preventing our core function – the googleRelationships function – from being used to support these extensions to our basic ontology builder. The

googleRelationships function itself is completely general, and to extend our current implementation to support these items would simply be a matter of doing more googleRelationships searches. For example, to add relationships between terms, we would simply call the googleRelationships function to perform searches between two terms after establishing these terms' basic relationship to the domain. Similarly, to support multiword terms, we would simply have to call googleRelationships on combinations of words from the potential terms pool in the initial ontology building step. The primary factor preventing us from doing these things is the associated cost – supporting any of these extra pieces of functionality would necessarily cause our number of required searches to explode, and hence would greatly increase the associated fee from Google. While this is at this time beyond our resources, in the future we may be able to acquire funding to further develop this algorithm and support this listed functionality.

There is an additional, much more important piece of future work which will be critical for an automatic ontology-building algorithm to be truly useful: this functionality is to make easier the task of defining which relationships the ontology-building algorithm is to build between its terms. The current GCKEngine requires us to manually program in how each relationship will use data from the googleRelationships algorithm to establish a relationship weight. This is a difficult task to do, and is certainly beyond the abilities of the average end-user. It is unlikely that we can find a way to automate this task, as manually configuring these relationships is both a crucial part of setting up the automatic ontology-building algorithm, as well as one of the core reasons that we are able to build an ontology automatically in the first place. Therefore, in future work it would be useful to find ways to facilitate this relationship building process – make it standardized, and easier to do. We do not want our GCKEngine to only be available to those who can program it – we want it to be usable by a common end user who wants the power of

semantic computing to organize his data. Therefore, making simpler the integration of relationships into an implementation of our engine is a necessary step to take in future work.

7. Conclusion

In this thesis, we proposed and developed an algorithm to automatically build an ontology for a certain domain. We were able to perform automatic ontology building by defining the relationships we will attempt to establish between the domain and the terms, and by using Google's search engine to establish whether a term has this relationship with the domain or not. To support this algorithm, we used a web crawler to download a series of documents, a classifier to extract the good documents, and a term extracting algorithm to establish a pool of potential terms. After running the algorithm and generating an ontology, we then used a tagging algorithm to apply the ontology tags to the documents we downloaded; we then displayed these documents in a Solr search engine.

We performed various tests on our algorithm: among these an analysis of the terms that make up the automatically-generated ontology, and a precision and recall test for each system we ran through the GCKEngine. These tests indicated that our ontology is a reasonably good one, and that we are able to effectively tag documents with it.

We built the GCKEngine in order to facilitate further development of the Semantic Web, and semantic computing as a whole. In order for semantic computing to truly move forward, we will need to find ways to automate the relatively complex task of building a strong ontology. We believe that our GCKEngine is a good step in this direction. At this early stage, it requires that its users to manually program the relationships it is meant to build into the ontology – but future developments should make this process easier to do.

In our tests of the engine, we used it to tag computer tutorials for three different systems, and this produced three sets of tagged documents of relatively high quality. While the quality may at this time not be sufficient for applications where high accuracy is mission-critical, it is more than enough for an average task such as searching documents like computer tutorials. The automatic ontology-building algorithm shows enormous promise, and it is our hope that this work may be continued in the future, to the end of bringing about the advancement of the Semantic Web.

References

Berners-Lee, Tim "The World Wide Web: A very short personal history" *W3.org*. May 7 1998. http://www.w3.org/People/Berners-Lee/ShortHistory.html

Berners-Lee, Tim, James Hendler, and Ora Lassila. "The semantic web." *Scientific american* 284.5 (2001): 28-37. http://www.scientificamerican.com/article.cfm?id=the-semantic-web

Shadbolt, N.; Hall, W.; Berners-Lee, T., "The Semantic Web Revisited," *Intelligent Systems, IEEE* , vol.21, no.3, pp.96,101, Jan.-Feb. 2006 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1637364&isnumber=34311

Bellandi, A.; Nasoni, S.; Tommasi, A.; Zavattari, C., "Ontology-Driven Relation Extraction by Pattern Discovery," *Information, Process, and Knowledge Management, 2010. eKNOW '10. Second International Conference on* , vol., no., pp.1,6, 10-15 Feb. 2010 doi: 10.1109/eKNOW.2010.17 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5430049&isnumber=5430020

Kalender, M.; Jiangbo Dang; Uskudarli, S.; , "Semantic TagPrint - Tagging and Indexing Content for Semantic Search and Content Management," Semantic Computing (ICSC), 2010 IEEE Fourth International Conference on , vol., no., pp.260-267, 22-24 Sept. 2010 doi: 10.1109/ICSC.2010.53 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5629261&isnumber=5628562

Ganter, Bernhard, and Gerd Stumme. "Creation and merging of ontology top-levels." *Conceptual Structures for Knowledge Creation and Communication*(2003): 131-145.

Knublauch, Holger. "An AI tool for the real world Knowledge modeling with Protégé." (2003). *JavaWorld*. http://www.javaworld.com/javaworld/jw-06-2003/jw-0620-protege.html