Automated Detection of Duplicate Free-Form English Bug Reports

Trevor C. Kemp

Problem report submitted to the
College of Engineering and Mineral Resources
West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Committee Members:
Dr. Bojan Cukic, Chair
Dr. James D. Mooney
Dr. Afzel Noore

Department of Computer Science

Morgantown, West Virginia
2009

Keywords: Natural Language Processing, Vector Space Product, Bug Report

Abstract

Automated Detection of Duplicate Free-Form English Bug Reports

Trevor C. Kemp

Every day, software developers receive reports on defects or suggestions for improvement in their products. A common way to relay this information is through a report collection tool called a bug report repository. For sufficiently large products, significant time can be consumed by personnel attempting to put bug reports through triage when similar reports addressing the same issue already exist in the repository. This report describes and experiment to automate this process by using natural language processing and a vector cosine technique applied to phrase matches found in an index of reports to score the likelihood that a bug report is a duplicate of another report. Since it can be difficult to determine an exact relationship between phrase matches and similarity between reports, a neural network is employed to learn the general relationship when reports are duplicates and when they are not in order to classify incoming reports appropriately.

Table of Contents

Abstract ii
Table of Contents iii
List of Figures and Tables v
Chapter 1: Introduction 1
Chapter 2: Related Work 2
Chapter 3: The Dataset 4
Gathering Data 4
Parsing the Reports 4
Chapter 4: The Experiment 6
Some Setup 6
Indexing Root Duplicates 7
Preprocessing the Descriptions 9
Stemming 9
Synonym Insertions 10
The Search and Scoring Process 11
Searching 11
Scoring 12
Creating Result Vectors 13
Using the Learner 13
About the Learner 13
Preparing Tests 14
Conducting Testing 16
Chapter 5: Results 16
Chapter 6: Discussion 18
Discussion of Results 18
Other Practical Limitations 19
Chapter 7: Conclusions and Future Work 19
Conclusions 19

Suggestions for Improving Run-Time 20
Exploiting Code Parallelism 20
Maintaining a Smaller Index 20
Processing the Shortest Phrases Differently 21
Improving Learner Success 21
Treating Phrases as Words 21
Refining Experiment Results 22
Rethinking the Repository 22
Bibliography 23
Appendix 24

List of Figures and Tables

Figure 1- An example bug report with underlying markup……5

Figure 2- Illustration of inverse duplicate table……...7

Figure 3- A simple example index……9

Table 1- Tabular depiction of test data……...14

Table 2- Results from each of 10 tests dissimilarity error method 1……17

Table 3- Results from each of 10 tests dissimilarity error method 2……17

Table 4- Global cross validation results dissimilarity error method 1……18

Table 5- Global cross validation results dissimilarity error method 2……18

Chapter 1: Introduction

As software developers author the projects they are involved with, it is inevitable that software defects will arise for any sufficiently large project. At times, these defects will arise during product testing before an official release is issued. However, since it is impossible to test a system in a manner that mirrors the exact way that system users will interact with it, a large number of defects are discovered by people other than those who designed and built the system. There exist tools which facilitate communication between developers on a project as well as provide a means for individuals who are not on the project to communicate concerns and report about errors they have discovered through normal use of the system. One such common tool is a bug repository such as BugZilla. BugZilla provides a mechanism to automate bug report collection by allowing users to enter their bug report information through a web interface. Collection is only one small part of a bug report's life, however.

Once a bug report exists in the repository, it goes through a process known as triage. During triage, the report is examined to garner information that developers will use to best resolve the problem. Each report contains information about the platform on which the defect was discovered, but the repository software also allows bug reporters to type in a free-form textual description of what happened when the defect arose. The true severity of the bug is determined, whether or not the bug will be fixed, which developer will fix it, and the bug report may be determined to be a duplicate of another report.

In the case of duplicate bug reports, it is highly desirable to filter these reports so that a developer does not need to expend time analyzing many reports that refer to the same defect. Additionally, it is desirable to automate this process in order to allow those performing triage to allocate their time more efficiently for the other triage tasks.

Hereafter, an experiment is described that reports the efficacy of combining minor changes to previously proposed natural language vector-cosine techniques for duplicate text identification with a machine learning technique that should discover what signature a duplicate of a report will have compared to the signature of a non-duplicate report. Only the free-form, reporter-entered text field from a bug report is used in this experiment to determine if it is a duplicate of another report. In implementation, this equates to building an index for the purpose of storing reports, which is later used to find common phrases with a new report. Search results are then preprocessed for training and testing on a well-known data mining package. The machine

learner that discovers what is a good candidate to be a duplicate of a new report and what is not a good candidate to be a duplicate is the Weka MultiLayerPerceptron module. The module is trained and tested with vectors from search results that contain the number of each phrases of a certain length (one element each in the vector for each length 1 to n, where n is the depth of the index) that an indexed report has in common with a new report, as well as the sum vector-cosine error from all phrases that two reports have in common. Using this technique, we can correctly identify 24% of the indexed reports as duplicates of a new report while incorrectly identifying 3.1% of indexed reports as duplicates of a new report, or correctly identify 14% of indexed reports as duplicates while incorrectly identifying 1.1% of indexed reports as duplicates using a slightly different method. Results from each method are validated with 10-way cross validation.

In the coming chapters, this experiment is explained in much greater detail. Chapter 2 will discuss related work by other authors. Chapter 3 will describe the dataset used in the experiment. Chapter 4 will describe the experiment in detail, with results in chapter 5. Chapter 6 will describe some practical limitations of the system. Chapter 7 will conclude the report and discuss recommendations for future work.

Chapter 2: Related Work

There have been several studies since some early important papers on text classification. Problems that fall under the domain of textual classification include determining if a new email is spam, classifying source code to determine which language it was written in, determining the author of a document, and many others. Some very successful classification schemes have been produced. These include Bayesian schemes. However, this experiment's type of text classification is a very different problem than the ones mentioned. For the above types of classification problems, membership of a given class is contingent upon what the text contains. There is something intrinsic to the text which allows one to assign membership. However, the two document classes (duplicate and non-duplicate) in this experiment have nothing intrinsic about individual reports which determine membership. Thus, identifying duplicate bug reports is actually a very different problem than normal text classification.

There has been much work in this particular brand of text classification. An excellent tutorial paper regarding the basics of natural language processing in relation to detecting duplicate bug reports using natural language processing has been written by Runeson, Alexandersson, and Nyholm (Runeson). It details the specific actions that must be taken to perform natural language processing on bug reports. They use a vector cosine model to determine similarity between reports. Their study report also contains results of many worthy smaller experiments to determine how decisions like adjusting time window of report collection, choosing a different similarity model, and other various items will affect the success of duplicate identification. Their experiment achieved a 30% recall (true positive) rate, but they also returned the 5 topmost likely articles to be duplicates of a new report. While there is much to learn from this paper, forcing developers to examine 5 articles for every new one that comes into the repository is probably not acceptable practice on most projects.

In his master's thesis, Lyndon Hiew describes a method for identifying duplicate bug reports by creating vectors for each report in a repository and grouping similar reports into *centroids* based upon their term-frequency-inverse document frequency, abbreviated tf-idf (Hiew). A centroid could be either a grouping of documents or a single unique document. These centroids have an associated vector. When a new report arrives, a tf-idf vector is created for it, and this vector is checked for similarity against all centroid vectors. The top centroid vectors are returned as candidate duplicates. With a 50% recall rate and a 30%precision rate, these were good results. However, as before, for each new report, triage personnel will need to examine several more reports.

Podgurski, et. Al. describe an experiment for classifying and grouping similar reports together (Podgurski). However, their data was quite different than in this experiment. They were classifying core dumps and captured program executions that had been submitted for 3 compiler projects. Using a clustering technique and multivariate visualization, they were able to classify and group similar failures based off of features that were gleaned from the reports by a regression technique. Similarity was determined by use of a distance metric computed for each cluster. The multivariate visualization was used to add a human opinion in the classification stage of the experiment because it was believed that automated clustering alone may not have achieved good results. As they refined each cluster that was built, clusters got smaller but had more similarity within each cluster. The result was that for the best cluster in each of their 3 datasets, each failure in that cluster was caused by the same thing, meaning 100% correct classification for this cluster.

Still another paper was written by Jalpert and Weimer (Jalpert) that describes a vector cosine technique combined with a clustering algorithm that will recognize 8% of duplicate new reports that arrive. The most important aspects of this experiment, however, were that the proposed system would have been implemented as an inline and online system. It would be inline because it would filter duplicate reports before they reached triage personnel, and it would be online because classification of one new report can be performed before new reports arrive.

Chapter 3: The Dataset

Gathering Data

The dataset consists of about 14,000 bug report files from The Eclipse Project (Eclipse Foundation), an open-source integrated development environment for many languages that is written in Java. Eclipse's hallmark is its ability to easily integrate plug-ins to give the application new functionality. It is maintained by the non-profit Eclipse Foundation, which makes its bug repository available publicly through anonymous (in this case meaning no login is required), encrypted HTTP connection. To download the bug reports, a script was written to call the underlying Linux shell command ***get***, which downloads an HTML page to file.

In order to use the ***get*** command, one must know the URL of the file to be downloaded. The Eclipse Foundation uses Bugzilla for its bug repository, which allows for users to access individual bug reports directly without navigating through the web interface. This is useful because one can automate the collection process. For instance, accessing the URL https://bugs.eclipse.org/bugs/show_bug.cgi?id=2001 will allow the user to download bug number 2001. To download all bug reports numbered between 2001 and 10,000, the script simply iterates from 2001 to 10,000, downloading each report.

Parsing the Reports

Each bug report is presented with fields displayed that show important information for the bug, like a description of the platform on which the bug occurred, the severity of the bug, the priority this bug has been given, a summary title, a textual description, and a status of either ***resolved*** or ***unresolved***. If the bug has been resolved, then there is an accompanying ***resolution*** that describes the findings or actions by developers for the bug. This resolution has some predefined options that triage personnel can choose to set after viewing the report. Of significant importance to this experiment is the ***duplicate*** option that triage personnel can set. The duplicate option will also display a bug report number of the report that has been duplicated, which can be parsed from HTML in order to download the duplicate.

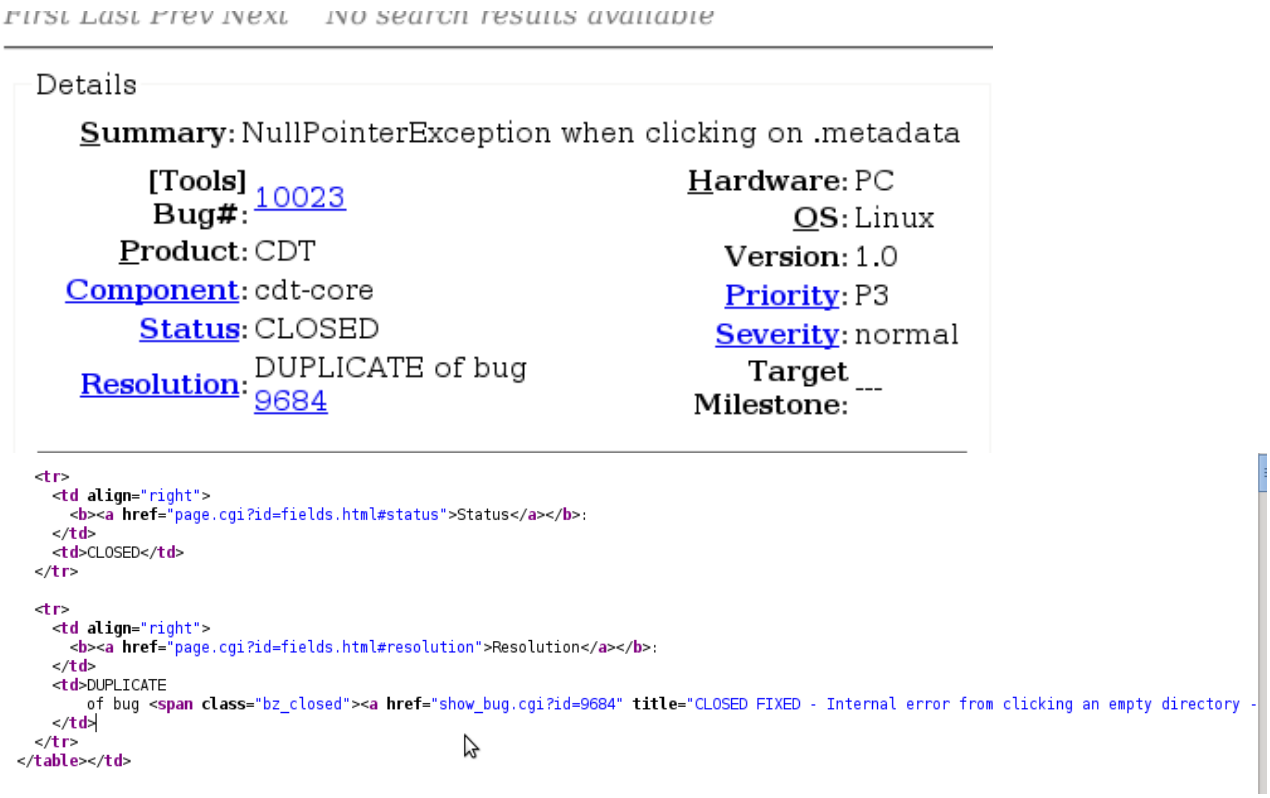

Figure 1- An example bug report with underlying markup. The HTML can be easily parsed to determine that this report duplicates the report numbered 9684.

Parsing the HTML files is simple. Because each bug report page is generated automatically, one only has to search for specific tags in the file to determine the bug report's resolution. If the resolution is marked as a duplicate of another report, then the URL to the original report (referred to here as the *root duplicate*) is also downloaded, but is isolated from other bug reports and placed with root duplicates of other reports.

To get the textual description of the bug report, the HTML is again searched for another specific sentinel text. Once this text has been found, the description is read into memory in its current form from the file, and a regular expression is used to strip out the residual HTML in the textual description. The result is a plain-text description of a bug report that is stored for use later, along with its root duplicate's report number if a root duplicate exists for this report.

In this experiment, other than using the resolution field to determine whether or not a report is a duplicate, only the textual description of these bug reports is used. It is noteworthy that the text itself in the report is rarely likely to be identical, and the errors may have occurred on different platforms altogether. Generally speaking, developers will use the semantics (expressed in

natural language) in the text field of the bug being described to determine if a new report is a duplicate of another bug. The other information contained in the report is merely supplementary. Additionally, this experiment searches on an index created solely from root duplicates. This allows for a shorter run-time without affecting experimental results, as discussed in subsequent sections.

Chapter 4: The Experiment

Some Setup

The experiment consists of several small tasks. Each part developed has been coded in Python. First, all bug reports, except for those in the isolated root duplicate folder, are indexed for experimental administration purposes. The index is a hash table that uses bug report numbers as keys. When a bug report is indexed, its full textual description is stored along with its root duplicate, as described in section 3. Because this hash table contains bug reports where each may or may not have a root duplicate, this hash table is referred to as the ***assorted reports*** hash table.

Another hash table, the ***inverse duplicate*** table, is also created to easily retrieve information regarding what reports a root duplicate belongs to. The name inverse comes from the idea of an inverted index where, for example, a web search engine will maintain what web pages a given text string appears on. For each item in the assorted reports table, if there exists a root duplicate, the current report number (for the assorted report) is appended to a list stored in the inverse duplicate table pointed to by the root duplicate's report number. If an assorted report does not have a root duplicate, its report number is appended to a list in the same hash table with an appropriate key to indicate that the items in the list are not duplicates.

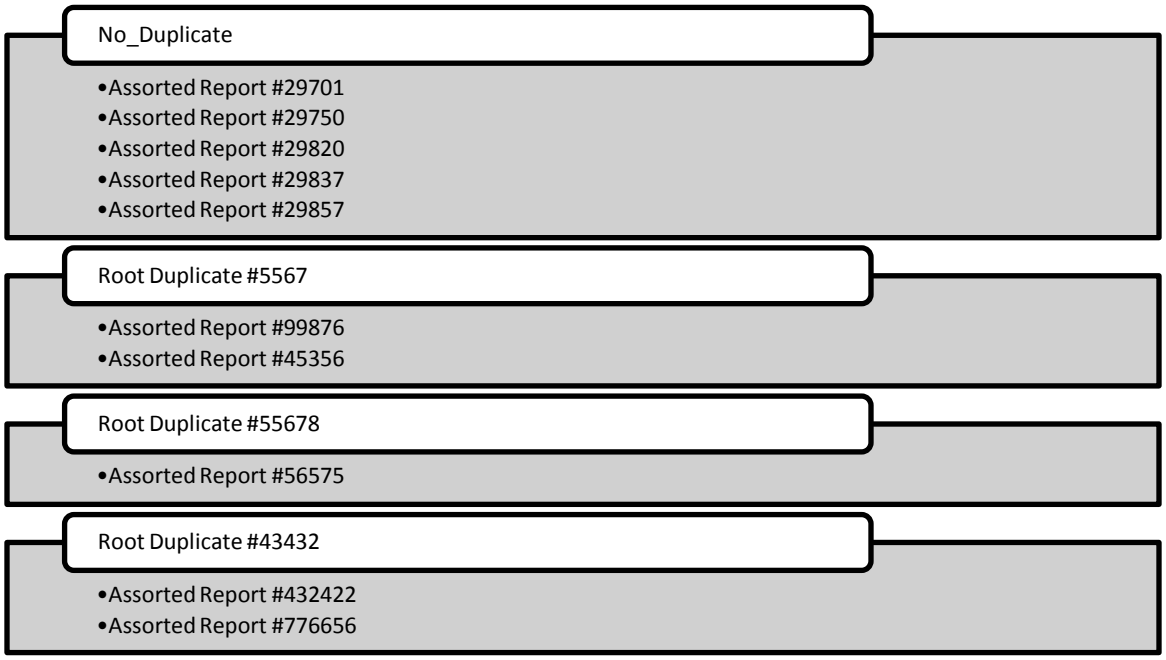

Figure 2- An example illustration of the Inverse Duplicate Table.

Indexing Root Duplicates

Once these hash tables have been populated, the root duplicates are indexed in the ***index*** object. It is the index that is searched against to uncover potential matches for a new article. In practice, all known bug reports would be indexed. However, for experimental purposes, only root duplicates are inserted. A bug report can bring no meaning to the word “duplicate” in and of itself. This is to say that since there is nothing intrinsic to a bug report that determines if it is a duplicate, it is the comparison between two reports that determines if the reports are duplicates of each other. With this in mind, we can safely insert only root duplicates into the index because when a report ***r*** returns matching words or phrases with a particular root duplicate ***rd*** from a search on the index, it makes no difference if these reports do not address the same issue because the machine learner is still informed during training and testing whether or not these reports are duplicates. Indexing only root duplicates allows for a faster running experiment without affecting the quality of the results used to train the learner or conduct tests.

The index should allow for one to search for entire common phrase matches between articles rather than just the existence of single common words. Indeed, the experiment is built on the premise that duplicate bug reports will share more phrases in common than non-duplicate reports even though reports of both types will contain many common words. In order to have the ability to search the index for phrases, a word-level suffix tree is built. In this case, a suffix is not

determined similar to those that exist in computational biology where each character in the text is the start of a new suffix. Instead, each word in a text is treated as one position in a suffix rather than each character being considered as a position in a suffix. The suffix tree thus operates at the word level, not the character level.

However, because the words to be indexed will come from several different reports, many of which will not exist until after the initial indexing, the index must be built similar to those constructed with online algorithms (constructed in linear time and built left-to-right from the text as opposed to being built right-to-left). However, such algorithms introduce an unnecessary complexity to the project. Linear insertion time and memory consumption can be achieved by limiting the depth of the index. In addition to giving a less complex indexing algorithm, informal timing estimates indicate that indexing reports in this fashion can be accomplished in an acceptable amount of time, with each report being indexed in less than half a minute on average. In this project, an index depth of 5 was chosen. This depth has proven sufficient, as it is rare for bug reports to have common phrases of this length. To build the index, a linked list is constructed where each successive node corresponds to a new position in a suffix from a bug report description. Each suffix is inserted starting at the root of the index. Each node in the linked list contains a hash table. The actual word that is to be indexed at this position in the linked list is used as they key. When a word is hashed, the term frequency and report number of the bug being indexed are stored. When the index is searched, the former item is used as a measure of similarity of usage for a word in two articles and the latter provides the mechanism for identifying the reports that have commonality with a new report. Each root duplicate is indexed in this manner.

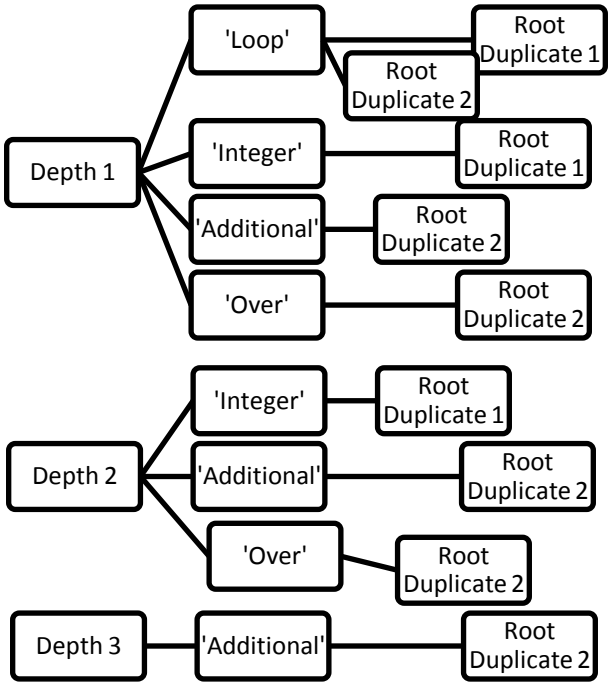

Figure 3- A simple example index. Root duplicate 1 would contain the text "Loop Integer" and root duplicate 2 would contain the text "Loop Over Additional."

Preprocessing the Descriptions

Stemming

Before the reports can be indexed, the descriptions must be preprocessed. The first preprocessing action is to stem the words. The purpose of stemming is to reduce the diversity in the index when words may have commonality and actually refer to the same thing. Stemming is the process of algorithmically determining the root of a word. Examples of stemming include reducing the word “running” to “run”, or “cable” to “cabl’. By stemming words, the index will return more and longer phrase matches when words may be referring to the same idea, but have different inflections. Since stemming makes similar words with different inflections reduce to a common word, it does not matter if the root word returned by the algorithm is actually a valid word at all. The important matter is accuracy; that if words share a common root, then the algorithm must return the same root for each of the words. The algorithm used is the popular Porter Stemming algorithm, provided by the Natural Language Toolkit (Natural Language Toolkit), an open source natural language processing package written entirely in Python. In addition to stemming the report words, non-alphabetic words are removed along with punctuation.

Synonym Insertions

Another issue when processing textual documents containing natural language is that very often, it is possible to describe the same event or scenario with different words that essentially refer to the same idea. Usually, one synonym of a word may be chosen over another synonym because it more accurately conveys the author's intent. This may because of pure definitional differences, or it may be due to connotations that words may carry with them. Regardless, when processing natural language in this application, we are left with the task of mitigating the effects introduced by authors using different synonyms when describing the same bug in different reports. There are two ways to accomplish this task, referred to here as ***synonym reduction*** and ***synonym indexing***.

Synonym reduction is the process of algorithmically discovering the commonalities in synonym usage in a given corpus or thesaurus. The point is to remove the subtle differences caused by authors' usage of different synonyms by reducing the number of synonyms in the corpus or thesaurus. It is accomplished by determining, for each set of words containing common synonyms, which word should stand in place of several other words used in a similar context. A simple means to accomplish this is to choose one synonym (random choice, choose the first word encountered each time, etc.) to stand in place of all synonyms that "link together" in a thesaurus. This is to say that if one were to group all words into sets based upon which words are synonyms of other, the result would be several disjoint sets, and each set would choose a representative so that whenever a word is encountered, its representative would be returned. The problem though is that when a word has a synonym, the word and its synonym are not perfect equivalences of each other. This simple method of synonym reduction yielded very poor results in the experiment and was abandoned.

The other method of coping with synonyms is to look the synonyms up in a thesaurus and to index the synonyms of a particular word alongside the original word or likewise instead performing the same functions at search time. This latter method sacrifices longer search times for shorter indexing times and a smaller index. In this experiment, the synonym indexing method was chosen and implemented by inserting the suffixes of a bug report as usual, but also inserting the synonyms of each word in the suffix at the same spot in the index as the original word. By this method, all known synonyms of a word will be searched for a match at no extra cost in search time. However, both of the methods mentioned for dealing with word synonyms assume the existence of a thesaurus.

A thesaurus of over 60,000 words was gathered from an online thesaurus. The thesaurus service was provided by words.bighugelabs.com (Watson). This website allows users to look up a word and download the synonyms of said word. An API is provided to accomplish this. For instance, the link http://words.bighugelabs.com/api/2/api_key/love/xml will return the synonyms for “love” in XML format, provided that the user has an API key that was issued by the website. The website allows 10,000 free requests per API key per day. A text file containing 300,000 words was used to seed the lookup process. Once all of the synonyms were downloaded, they were stemmed and indexed in hash tables, using one hash table to store thesaurus entry words, and several other hash tables to store the entries’ synonyms, with each entry in the thesaurus pointing to the hash table that contains its synonyms. This allows for fast synonym lookup.

The Search and Scoring Process

The purpose of the search and scoring process is to prepare the articles for the machine learner that is used to classify reports as duplicates of each other or not duplicates of each other. Searching and scoring are accomplished by way of several small steps.

Searching

First, the report’s words are stemmed. Punctuation is removed. Non-alphabetic words are removed. Then, the suffixes of the new article are successively searched for in the index. The search results that are returned contain common phrases between the new article and the articles in the index. In addition to this, for each common word, the program maintains the term frequency of the words in both the new report and the indexed reports. The term frequency is directly used to calculate the similarity between two reports. It is also notable that if an index returns a phrase match of length ***n*** for a suffix ***s*** in the new report, then the indexed report will be ignored until we are searching for the new report’s ***n+s-1*** suffix. This prevents bloating the search results unnecessarily and will also help the learner to distinguish a duplicate from a non-duplicate.

Importantly, it was noted that because this is real-life data, some of the assorted reports are also root duplicates. In order to guarantee a sanitary test case, if we begin searching with a report and get phrase matches back from the same report because it is indexed as a root duplicate, we do not use these results in the test cases. They are removed from the result set entirely; for just one search, it is as though the root duplicate is not indexed at all.

Scoring

The search results are arranged into vectors. For each common phrase, there are two vectors. One vector corresponds to the term frequency of the new report's words from the phrase. Jalpert's formula of *1+log(term frequency)* is used to reduce the effects of relatively small differences in term frequencies in the two reports. The second vector corresponds to the term frequency of the indexed report's words from the phrase. Then, the vector cosine

$$\frac{a \cdot b}{\| a \| * \| b \|}$$

is computed and stored, where ***a*** is the new report's vector and ***b*** is the indexed report's vector. The closer the cosine value is to 0, the more similar was the use of the phrase in the two reports. The error in phrase usage similarity is calculated as *1-abs(cosine(a,b))*. In addition to these calculations, if the length of the common phrase is greater than 1, the cosine for this phrase is divided by *log(length)* in order to give a reward for having a longer phrase in common. Note that it is possible, for each phrase, to compute these term frequencies at the phrase level rather than the word level to get a more accurate cosine error score.

At this point, each indexed article that returned search results will likely have several such cosine values. These cosine values are summed to produce a ***similarity error***. However, since this similarity error only takes common phrases into account, reports that have very much in common with the new report are penalized much more heavily than those that have less in common with the new report. In order to penalize reports with less in common, another type of error is calculated, the ***dissimilarity error*** (Lemon 09). There were two methods explored for computing this error.

For the first method, the error is calculated by finding the number of words from *a* that do not appear in *b* and then dividing the length of *a* by this value. Since the length of *a* will be greater than the number of words from *a* that do not appear in *b*, the resulting improper fraction will be greater than 1 (since there is a search result returned, the articles have some commonality and such will never return a fraction equal to 1).We then take the *log* of this value and get its reciprocal. This reciprocal value is the dissimilarity error.

The second method of error calculation is simpler. We simply divide the number of words in *a* that do no appear in *b* and then divide this number by the length of *a*. This yields the percentage of report *a* that does not appear in *b*. We take the *log* of this value in order to slow the growth.

The first method essentially progressively penalizes reports that have more than half of the words uncommon while increasingly rewarding reports that have more than half of the words in common. The second method is a logarithmically increasing penalty; the penalty increases slowly as the number of uncommon words increases and never rewards for having reports having little uncommon.

Once these calculations have been performed using either method, the similarity error and the dissimilarity error are multiplied together to get the ***total error*** for the two reports in question.

Creating Result Vectors

Once the search has been performed and scored, it is time to arrange the results into suitable input for the learner. To do this, a ***result vector*** of length ***d + 2***, where *d* is the depth of the index, is created. The first *i..d* entries correspond to the count of phrase matches of length *i* for the indexed report. The *d+1* element of the result vector is the total error for the report. The *d+2* element is a textual string that tells the learner whether or not this vector corresponds to a duplicate or a non-duplicate. The duplicate/non-duplicate status is known for this experiment. This field is used only for training and evaluation. An example result vector for a non-duplicate searched against an index depth of 3, 2 common phrases of length 1, 2 common phrases of length 2, 0 common phrases of length 3, a and total error of 0.00223 would take the form

<2, 2, 0, 0.00223, 'no_dup'>

Each of the first *d* elements in the array is normalized to 1. The final result vector would take the form

<0.5, 0.5, 0, 0.00223, 'no_dup'>

Each indexed report that returns search results will have such a vector. At this point, the scoring process is complete and each element in the vector is used as an input to the learner.

Using the Learner

About the Learner

The learner used was the MultilayerPerceptron functionality provided in the Weka data mining package (Frank). Such a neural network is more appropriate than other learners like Bayesian approaches because the neural network is capable of generalizing relationships and classifying based off of inputs that it has never seen before (perfect for floating point calculations), whereas a Bayesian approach requires some subset of symbols in the inputs to have been seen before.

For this experiment, since the inputs will always be floating point numbers that can vary greatly, the Bayesian approach is inappropriate, although it is simpler to use and understand.

The MultilayerPerceptron function in Weka is a feed-forward network, meaning that there are no cycles in the network topology. For each input processed, the same artificial neuron will never be visited twice. Additionally, this package uses back propagation to correct error while training. The package allows a user to determine the network's training parameters. For this experiment, trial-and-error produced acceptable experimental results with the following settings: *random seed* = 13, *hiddenlayers* = a (automatic construction by Weka), *learning rate* = 0.5, *momentum* = 0.2, *training time* = 5000 or 8000 epochs, depending on which dissimilarity error computation method was used (5000 for method 1, 8000 for method 2).

Preparing Tests

The tests were performed with 10-way cross validation. The data is partitioned as explained. Since there are slightly more than 1600 root duplicates, each run of the cross validation contains 160 reports from the assorted reports dictionary that have a root duplicate. These are selected by randomly choosing a key from the inverse duplicate table and then choosing a report that is a duplicate of this key. This report is removed from the assorted reports table and also from the root duplicate table. If the root duplicate key has no more reports listed as duplicates, the key entry is removed from the table entirely. In addition to 160 assorted reports with duplicates, 160 assorted reports are chosen at random and removed from the assorted reports table. This grouping of 320 articles is called a ***data bin***. When choosing assorted reports at random, assorted reports that have a root duplicate are not chosen. In this manner, no report will appear in more than 1 time and in no more than 1 data bin.

Data Bin	
Reports With Duplicates	160
Reports With No Duplicates	160
Total Reports Per Data Bin	320

Test Set (Search Results from a Data Bin)	
Duplicate Result Vectors	160
Non-Duplicate Result Vectors	500,000+
Total Test Cases	500,160+

Training Set (Subset of a Test Set)	
Duplicate Result Vectors	160
Non-Duplicate Result Vectors	1440
Total Training Cases	1600

Table 1- A tabular depiction of how experimental data is constructed.

Once this has been completed, the reports in each data bin are searched against the index. The result vectors from this searching and scoring are written to a ***test set***. There are 10 such test sets, one for each data bin. Results show that after performing the search, each test set contains 160 result vectors classified as duplicates and 3-4 thousand times that number of result vectors classified as non-duplicates. The large difference in the amount of duplicate vectors to non-duplicate vectors is because each time article is searched against the index, any article with commonality will return a vector. Since each data bin contains only 160 duplicates, this is the maximum number of duplicate vectors that can appear. However, every indexed report that had commonality with any report in this data bin that also happens not to be a duplicate is in this set as well. With over 1600 reports indexed, and 320 reports to search in each data bin, this will result in about half of a million result vectors per test set.

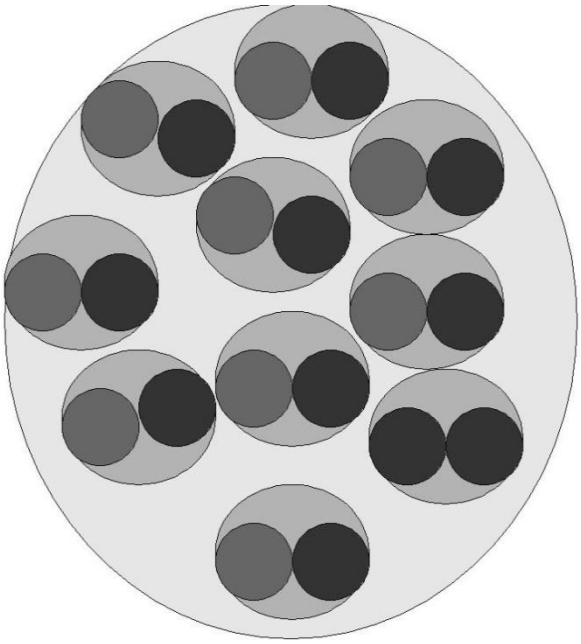

Figure 4- A diagram showing the construction of 10 data bins. Each cluster represents a data bin. The lighter circle in a cluster represents 160 reports with known duplicates. The darker circle represents 160 other reports. There is no overlap in datasets.

To train the learner, smaller ***training sets*** were constructed from each test set. The training cases for a single training set all come from the same test set of result vectors, and there is no overlap between training sets. Trial-and-error showed that the best results are yielded when the

test sets contain all result vectors classified as duplicates from a test set (160 test cases), and random result vectors that are not duplicates are chosen from the same test set until the training set is composed of about 10% result vectors classified as duplicates and the rest classified as non-duplicates for a total of about 1600 test cases in each training set.

Conducting Testing

To test, the 1st training set is used to train the learner. Then, the 9 test sets that were not used to create the 1st training set are used for testing. Results are recorded. The neural network is reset. The training process is repeated with the second training set. Each of the 9 test sets that were not used to create the 2nd training set are used to test. The results are recorded. This process is repeated until all 10 training sets have been used, and 90 tests have been conducted. In this manner, the results are produced and verified using 10-way cross validation.

Chapter 5: Results

Once the results were gathered for each dissimilarity error method, the mean true positive rate and the mean false positive rate were gathered from each of the 10 tests on training sets. For the first dissimilarity error method, some tests performed significantly higher (5% – 15%) than the mean true positive rate while scoring above the mean false positive rate only slightly (2.5%). For second method, the true positive rate remained around 14% while the false positive rate remained at about 1.1% in all but the most extreme cases. Both of the terms true positive and false positive are defined subsequently.

Test 1			**Test 2**	
Mean TP	0.285		Mean TP	0.236
Mean FP	0.039		Mean FP	0.027
Test 3			**Test 4**	
Mean TP	0.281		Mean TP	0.315
Mean FP	0.038		Mean FP	0.041
Test 5			**Test 6**	
Mean TP	0.285		Mean TP	0.193
Mean FP	0.038		Mean FP	0.020
Test 7			**Test 8**	
Mean TP	0.325		Mean TP	0.228
Mean FP	0.048		Mean FP	0.026
Test 9			**Test 10**	
Mean TP	0.091		Mean TP	0.223
Mean FP	0.005		Mean FP	0.027

Table 2- The values obtained from each test under dissimilarity error method 1. TP stands for True Positive. FP stands for False Positive.

Test 1			**Test 2**	
Mean TP	0.195		Mean TP	0.123
Mean FP	0.023		Mean FP	0.010
Test 3			**Test 4**	
Mean TP	0.165		Mean TP	0.144
Mean FP	0.011		Mean FP	0.006
Test 5			**Test 6**	
Mean TP	0.149		Mean TP	0.157
Mean FP	0.013		Mean FP	0.013
Test 7			**Test 8**	
Mean TP	0.098		Mean TP	0.067
Mean FP	0.004		Mean FP	0.002
Test 9			**Test 10**	
Mean TP	0.173		Mean TP	0.166
Mean FP	0.014		Mean FP	0.017

Table 3- Results from dissimilarity error method 2

In Figures 5 and 6, the results are listed. The true positive number corresponds to the fraction of result vectors that belong to reports that that have a duplicate and the vectors were also

correctly classified as characteristic of duplicates. The false positive corresponds to the fraction of result vectors that do not belong to reports with duplicates that were incorrectly classified as being characteristic of a duplicate. Once these figures were obtained, a global mean of both true positive percentages and false positive percentages was calculated, as was the variance in the test figures. The cross validation results are listed subsequently.

Mean TP	0.24667	**Variance TP**	0.00480
Mean FP	0.03126	**Variance FP**	0.00016

Table 4- Global mean and variance for dissimilarity error method 1

Mean TP	0.14426	**Variance TP**	0.00145
Mean FP	0.01164	**Variance FP**	0.00004

Table 5- Global mean and variance for dissimilarity error method 2

The cross validation reveals that the tests were highly consistent. The learner is capable of using the proposed model to correctly classify about 24.5% of duplicates with an expense of about 3.1% of non-duplicate search results being incorrectly classified, or to classify 14% of duplicates correctly at an expense of 1.1% of non-duplicate search results being classified incorrectly.

Chapter 6: Discussion

Discussion of Results

Although a true positive rate of 24.5% would be gladly welcomed by any project manager who has to manually examine bug reports in order to detect duplicates, this success rate comes at a very high cost in misidentification. Although it seems good that the false positive rate was so low even though the only information from the reports used were textual descriptions written in natural language, the 3.1% rate of false positives can be misleading. Since it refers to the percentage of result vectors from non-duplicate articles that were classified as duplicates, and each search yields one result vector for each indexed report that has any commonality at all with a new report, this means that for every new report that was searched, roughly 50 (about 520,000 result vectors per test set /320 articles for each data bin * 3% = 48.75) reports from the index will incorrectly be considered as a duplicate of the new report. Since each bug report cannot have more than one duplicate, these results are simply not usable in a practical system

because it adds too much overhead in triage. This number shrinks to about 18 duplicates per new report if the second dissimilarity error method is chosen. However, as the index size increases, so will the number of false duplicates per new report since more reports are likely to introduce more commonality.

Other Practical Limitations

The system was built as an experiment in combining machine learning with natural language techniques. This required a more granular approach to using the vector cosine technique in order to better train the learner. However, the sheer amount of data produced by a search on a full index of bug reports would be prohibitive to actively using the system. For every article that has a word or phrase in common with a new report, a result is returned. There is a high likelihood that most reports in the index will share some subset of words.

As the number of indexed articles grows, so will the search time. The run-time of a production-quality version of the system would then be $O(l* n)$, where l is the length of the new report, and n is the number of indexed reports. For small indexes (when l approaches n), this run-time is quadratic with respect to the number reports in the index. For large indexes, it results in an extremely slow sub-polynomial algorithm. In all but the best of cases, the algorithm will also result in producing an extremely large memory footprint that is not feasible for most systems to handle. Although the worst cases are not likely to happen, actual use of the system is very slow. This analysis corresponds to an implementation in a real system, not accounting for all operations to ensure a good experiment.

Additionally, in contrast to more slow parts of the system, it is noteworthy that in experimentation the learner added no significant overhead in clock time. For each test, hundreds of thousands of result vectors were classified within seconds. With a much larger index, this may change.

Chapter 7: Conclusions and Future Work

Conclusions

A system to automate the detection of some software bug reports was proposed that uses a granular, phrase-based vector cosine technique to prepare a neural network for classifying new bug reports as either a duplicate of an indexed bug report, or else as not a duplicate. An experiment was presented, along with results that demonstrate 24.5% of duplicates being detected at an expense of 3.1% false positives, or 14% being detected at an expense of 1.1% false positives under a slightly different method. The experiment shows that a neural network

can successfully be employed to recognize the general signature that a duplicate relationship between bug reports has. Practical limitations of the system are explained subsequently, and some methods for overcoming those limitations are also suggested. Additionally, suggestions for future work are presented in the form of new ideas to explore in an attempt to get better results from a machine learner. The experiment shows promise of better results in the application of older natural language computational techniques to a new problem.

Suggestions for Improving Run-Time

Exploiting Code Parallelism

There are some items that can be implemented on the system in order to improve the speed (in clock time) that the system performs a search and scores the results. First and simplest, the very platform where the work is performed has a lot to do with the amount of time it will take to compute the results. Although it is a very simple improvement that makes no change in theoretic run-time, the algorithm is ripe for parallelization. It would be a rather simple task to distribute the index across several computers, allowing each machine to perform part of the search. If the search on each machine is distributed among one or more processors (or processor cores), a substantial speedup can be expected since the search is a read-only process.

The scoring aspect of the experiment can also expect speedup from parallelism. By using clever orders of processing and arranging search results properly, almost every part of the scoring process can take advantage of inexpensive and efficient vector processors available today. These can come in the form of processors like the Cell B.E. or graphics processors that have available API's for scientific programming. Since the vectors in these processors are usually large, and there are a number of them available for use, this can reduce the run-time significantly even if it is only a scalar reduction. Although parallel algorithms can become very difficult to write, the task is almost trivial when there is not much data to be shared among running processes. In the case of scoring results with the vector cosine technique, the longest-running part of the algorithm would be preparing the results to maximize usage of the vectors in such a processor. The combination of these two types of inexpensively implemented parallelism can greatly reduce run-time without much of an overhead in algorithm construction.

Maintaining a Smaller Index

The size of results returned by a search directly affects the time it will take to classify a new report. By keeping the index small, fewer results will be returned. Other studies have indicated that new bug reports are likely to be duplicates of those that have also been created relatively

recently. An empirical study would best determine, for each project, the oldest reports that should be indexed. Since the purpose of the experiment is to reduce the time it takes to triage duplicates, not to eliminate it, it is acceptable to maintain a smaller index at the expense of missing some duplicate bug reports from older items. This would also likely reduce the false positive rate.

Processing the Shortest Phrases Differently

There is a large likelihood that since all bug reports for a particular project are referring to the same product that if any new report is searched that a lot of very short phrases will be matched from otherwise very different reports in the index. This will bloat the search results and greatly increase scoring time. One option is to provide an empirically-derived threshold in which articles with no longer phrase matches are only scored if they contain as many or more short matches as the required threshold, or perhaps ignoring the shorter phrases altogether until a minimum threshold of word matches exist for a given pair of articles. The definitions of “short match” and “long match” will also likely depend on the specific project for which bugs are being reported, and what data is being collected. There are likely many more ways to process shorter phrases differently, but the key is to reduce the work load created by many small, insignificant matches.

Improving Learner Success

The goal in improving the learner’s ability to classify documents should be to reduce the false positive rate without significantly lowering the true positive rate rather than aiming to significantly raise the true positive rate. There are a few ideas that can, perhaps, increase the classification success.

Treating Phrases as Words

The vector-cosine technique described calculates on several vectors based on a word-by-word basis for each vector. If one were to instead treat each common phrase between two articles as a single word, the phrases could be compared based upon their phrase frequency. The new report and the indexed report with commonality would arrange one large vector of these phrase frequencies, with each element of the vector corresponding to the frequency of a phrase’s appearance inside the report. Then, the vector cosine could be applied to find the error between the two reports. This information would then be used as the final numerical input to the learner rather than the sum of all error from phrase lengths. This technique would likely also provide a faster run-time, as only one vector cosine needs to be calculated per report in the search results.

Refining Experiment Results

Since often times declaring a bug report as a duplicate is a very political process on message boards, and some reports classified as duplicates actually bear little common language, it is possible that the learner is generalizing its knowledge in order to include these “difficult duplicates” and introducing many false positives in the process since non-duplicates can look similar to these particular duplicates. It would likely be advantageous to run another experiment after running it once, using just the correctly classified result vectors to train a new neural network and removing all incorrectly classified result vectors from the training set. By doing this, it is possible that the learner will be able to maintain the same 25% true positive rate and significantly lower the false positive rate by learning from correct examples only.

Rethinking the Repository

Perhaps the most problematic aspect of automatic bug report processing is that a user is free to enter a textual description he or she deems necessary. Although this is the very strength of a bug reporting tool, this can lead to a lot of confusion for an automating tool that relies on word frequencies, length of postings, phrase matches, and any other form of natural language processing to aid in identifying duplicate bug reports..

Some individuals write very accurate and concise bug reports. Others will write a description of the problem, and then proceed to issue a multi-paragraph complaint, masking the most useful parts of the description behind “noise.” Still other bug reports contain example output from applications or compiler warnings, or stack traces from a program crash. It is probable that the learner’s accuracy would be improved if bug repositories migrated to a more granular storage schema whereby users are instructed to include only a description of the problem in one field, code examples in another, stack traces from program crashes in another if applicable, and personal comments in yet another. It may be necessary to enforce these measures by restricting the sizes of certain fields. However, the most important aspect of this plan is to isolate key report description parts into their own group in a bug report form. This granularity would allow for a much greater degree of flexibility in automating the triage and duplicate detection processes in addition to reducing the amount of “noise” in a textual description.

Bibliography

Eclipse Foundation. Eclipse bugs. 2007. March,April 2009 <https://bugs.eclipse.org/bugs/>.

Frank, Ian H. Witten and Eibe. "Data Mining: Practical machine learning tools and techniques", 2nd Edition. San Francisco: Morgan Kaufmann, 2005.

Hiew, Lyndon. "Assisted Detection of Duplicate Bug Reports." MS Thesis University of British Columbia, May 2006. 19 April 2009 <www.cs.ubc.ca/grads/resources/thesis/Nov06/Hiew_Lyndon.pdf>.

Jalpert, Nicholas and Weimer, Westley. "Automated Duplicate Detection for Bug Tracking Systems." International Conference on Dependable Systems & Networks. Anchorage: IEEE, 2008. 52-61.

Lemon, Bryan. Personal Communication, February - March 2009.

Lewis, David and Ringuette, Marc. "A Comparison of Two Learning Algorithms for Text." 1994. CiteSeerx Beta. 19 April 2009 <http://www.cs.cmu.edu/~mnr/papers/categ.ps>.

Madnani, Nitin. "Getting Started on Natural Language Processing with Python." 2007. Madnani, Nitin Personal Site. 19 April 2009 <http://www.umiacs.umd.edu/~nmadnani/>.

Natural Language Toolkit. December 2008 <http://code.google.com/p/nltk/>.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J. and Wang, B. "Automated support for classifying software failure reports." 25th International Conference on Software Engineering. Portland, 2003. 465-475.

Runeson, Alexandersson and Nyholm. "Detection of Duplicate Defect Reports Using Natural Language Processing." 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007. 499-510.

Watson, John. Big Huge Thesaurus: Synonyms, antonyms, and rhymes (oh my!). 2009. January 2009 <http://words.bighugelabs.com>.

Appendix

Six sample bug reports are included. Each pair of two reports has been classified as duplicates. The first report is the original, the second is the duplicate. The data presented in the first two duplicate pairs illustrates why automated duplicate analysis in free-form text can be so difficult. It can be difficult for even a human to determine, based off of the description text alone that the reports should be considered duplicates.

Each page contains one report, with only minor editing to reduce the physical size and was copied directly from the HTML rendering in a browser. Each duplicate pair is separated by a heading for clear demarcation. These reports come from the Eclipse bug report site, https://bugs.eclipse.org/bugs/.

First Pair

Details

Summary: Cannot use CVS keywords in code templates

[Eclipse] Bug#: 27301

Product: JDT

Component: Core

Status: RESOLVED

Resolution: FIXED

Hardware: PC

OS: Windows 2000

Version: 2.1

Priority: P3

Severity: normal

Target Milestone: 2.1 M4

People

Reporter: pgawron@dmcs.p.lodz.pl

Assigned To: JDT-Core-Inbox <jdt-core-inbox@eclipse.org>

QA Contact:

Description:

Opened: 2002-11-28 07:44 -0400

```
I wanted to generate my own file template which would already use $Revision$
and $Log$ keywords. $ sign starts Eclipse variables in templates and I
receive
error "template has incomplete variables".

How to produce:
Menu: Window->Preferences->Java->Templates.
Choose template and edit template. Try to insert $Revision$.
I cannot confirm my new template.

Bye,
Piotr
```

Details

Summary: Problems starting Eclipse Linux-Motif

[Eclipse] Bug#:	27666	**Hardware**:	PC
Product:	Platform	**OS**:	Linux-Motif
Component:	SWT	**Version**:	2.1
Status:	RESOLVED	**Priority**:	P3
Resolution:	DUPLICATE of bug 27031	**Severity**:	normal
		Target Milestone:	---

People

Reporter: DJ Houghton <dj_houghton@ca.ibm.com>

Assigned To: Platform-SWT-Inbox <platform-swt-inbox@eclipse.org>

QA Contact:

Description:

Opened: 2002-12-04 11:05 -0400

```
build 2002-11-27

I have a new install of RedHat 8.0.
I downloaded and tried to start the motif build.
I get the following error message:

bash-2.05b$ Warning: Missing charsets in String to FontSet conversion
X Error of failed request:  BadFont (invalid Font parameter)
  Major opcode of failed request:  55 (X_CreateGC)
  Resource id in failed request:  0x1e00028
  Serial number of failed request:  297
  Current serial number in output stream:  337

Is there something wrong with my install?
Thanks.
```

Second Pair

Details

Summary: Tagging warns about uncomitted changes that are not visib...

[Eclipse] Bug#: 29026

Hardware: PC

Product: Platform

OS: Windows 2000

Component: Team

Version: 2.1

Status: RESOLVED

Priority: P3

Resolution: FIXED

Severity: normal

Target Milestone: 2.1 M5

People

Reporter: Øyvind Harboe <oyvind.harboe@zylin.com>

Assigned To: Michael Valenta <Michael_Valenta@ca.ibm.com>

QA Contact:

CC: flyguy@null.net
pascal_rapicault@ca.ibm.com

Description:

Opened: 2003-01-06 06:49 -0400

```
Here is what I did:

- Nobody else is using the repository

- Team->Synchronize. Result:

"Workspace resources are the same as remote".

- Team->Tag as version. Result:

"You are tagging 'foobar' that has uncomitted changes..."
```

Details

Summary:	Decorators show directory has updated files, replace w la...
[Eclipse] Bug#:	29413
Product:	Platform
Component:	Team
Status:	RESOLVED
Resolution:	DUPLICATE of bug 29026
Hardware:	PC
OS:	Windows 2000
Version:	2.1
Priority:	P3
Severity:	normal
Target Milestone:	---

People

Reporter:	IH <flyguy@null.net>
Assigned To:	Platform-VCM-Inbox <platform-vcm-inbox@eclipse.org>
QA Contact:	

Description: **Opened:** 2003-01-13 16:35 -0400

```
Decorators show directory has updated files, replace w latest doesn't remove
decorator.  No files are highlighted in the directory.  Synch with repository
reports same as source.

cvs update -C -d -P "/directory"

I turned off prune empty directories and then issued the replace with latest.
The decorator turned off and a directory was created with no files inside.  I
then turned prune empty directories back on and replace with latest.  The
directory was pruned and the decorator remained off.

This is using M4
```

Third Pair

Details

Summary: [Key Bindings] Keybinding preference page: Usability

[Eclipse] Bug#: 28470

Hardware: PC

Product: Platform

OS: Windows 2000

Component: UI

Version: 2.1

Status: RESOLVED

Priority: P3

Resolution: FIXED

Severity: major

Target Milestone: ---

People

Reporter: Martin Aeschlimann <martin_aeschlimann@ch.ibm.com>

Assigned To: Chris McLaren <csmclaren@andelain.com>

QA Contact:

CC: akiezun@mit.edu

Description:

Opened: 2002-12-17 05:55 -0400

```
20021216

The current configuration dialog for keybindings requires a lot of clicks
through combo boxes to get things done. Especially when you want to get a
global picture of where is what action configured, this requires to go
through
each combination.

I was thinking that a table of all shortcurts with there currently assigned
action would be a good approach.

Having a combo box to select which configuration should be changed
(Default/Emacs) and the table with columns
'shortcut', 'global action' and 'text action'
where the table contains all possible shortcuts with the current action for
the
selected configuration.
```

Details

Summary: key bindings - UI is the other way around

[Eclipse] Bug#: 28493	**Hardware**: PC
Product: Platform	**OS**: Linux-Motif
Component: UI	**Version**: 2.1
Status: RESOLVED	**Priority**: P3
Resolution: DUPLICATE of bug 28470	**Severity**: major
	Target Milestone: ---

People

Reporter: Adam Kiezun <akiezun@mit.edu>

Assigned To: Platform-UI-Inbox <Platform-UI-Inbox@eclipse.org>

QA Contact:

Description:

Opened: 2002-12-17 08:12 -0400

```
20021216
there's no easy way to do the most obvious thing that people will want to do
-
namely, change the binding for a function

as it is now, you have to know the key sequence first.
i never do - if i did i wouldn't be chaning the binding - i know what the
function is, however

major - makes the feature half-usable only
```