# Probabilistic Model for Classifying Commits to Linux Memory Management Source Code

Tharun Medini
Electrical and Computer Engineering
Rice University

Sai Madem
Electrical and Computer Engineering
Rice University

## Abstract

We propose a Natural Language Processing based model to predict whether a commit to linux memory management source code falls into the class of *bug fixes* or *code maintenance* or *optimization* or *new features*. This is useful when we want to study a particular class of commits and analyze/locate a particular trend/feature. While bug fixes mostly have the keyword *fix* to identify, it is not the same for other classes of commits like *optimization*. A byproduct of this tool is to identify the keywords used for each class of commits.

## 1. Methodology

We used the plugin *webscraper.io* to traverse through the GitHub repository of Linux kernel source code and to create a database of the latest commits. This is done by creating a site-map of the kernel's memory management subsystem. Database of the commits is generated by collecting information like title of the commit, description of the commit, date of the commit and the file to which the commit is made. After generating the database we labeled the latest 977 commits as to whether they correspond to a 'bug fix' or 'code maintenance' or 'optimization' or 'new feature' by studying the commit title and description. Of the labeled 977 commits, 800 are used to train the model and the rest 177 are reserved for testing the accuracy. This partition follows the heuristic 80:20 division of data. The model was trained using only the short description first and later it was trained using both short and full description of the commits. We noticed that just using short description was giving slightly better results, suggestive of the fact that linux community uses technical jargon that may be inconsistent from person to person and hence the full description is not adding much information. Here are the brief steps that we perform in our algorithm.



### 1.1 Data Pre-processing

We first split the descriptions by using *.split()* function in python and then create a dictionary(equivalent to hashmap in Java) of all words seen across all commit descriptions. We call it *lookup*. While it is customary to omit numeric and special characters, we used them because certain terms like *use-after-free* are consequential to determining the class of a commit. Also, we converted all characters to small case in order not to have duplication of words. The dictionary contains all unigrams(each word is a unigram) and bigrams(each consecutive pair of words is a bigram). After completion of dictionary formation, we assign an index to every unigram and bigram present in the the lookup. We had a total of 3116 entries in the lookup.

### 1.2 Featurization

For every commit, we need to generate a feature vector to plug into a classifier. We generate sparse high dimensional(3116 dimensional) features by using indicator function for all terms present in a description by *'looking up from lookup'*. This gives all column numbers where we need to have *1* and the rest of the columns by default have *0*. We use a python sparse data structure called $csr_matrix$ to efficiently represent and compute the features of all commits.

### 1.3 Classifier

We tried a myriad of classifiers like Support Vector Machines, Logistic Regression, Random Forests etc. Among all, Logistic Regression with *'sag'* with a inverse regularization parameter of $10$ seems to perform the best, suggestive of the fact that a regularization coefficient of $0.1$ is appropriate for this data and task.

After training with the chosen classifier and features mentioned above, we can check the accuracy on test data(177 commits that we reserved for testing) and report how our model performs.

## 2. Results and Observations

We generate sparse features from descriptions of commits as mentioned in the Methodology section. We then trained the model using the short description of the commits. As mentioned earlier we got better results with just the short description of the commits. The outline of our results is as follows
Overall training accuracy - 96.33%
Overall testing accuracy - 61.14%
Accuracy on Bug fixes - 62%
Accuracy on new features - 56%
Accuracy on Optimization - 46%
Accuracy on maintenance - 76%

### 2.1 Interesting Points

Sanity Check with top keywords in each category:
*Bug fixes* - fix, remove, check, avoid, compound_head(), robust
*Code Maintenance* - rename, replace, unexport, constify
*Optimization* - remove, unnecessary, reference_manipulation, update, page_reference, compaction
*New Features* - add, introduce, support, implement

Sample of wrongly classified cases:
*Bug Fixe*s - replace kswapd compaction with waking up kcompactd, reintroduce split_huge_page()
*Code Maintenance* - compaction: distinguish between full and partial COMPACT_COMPLETE
*Optimization* - use page_cgroup_ino for filtering by memcg, avoid looking up the first zone in a zonelist twice
*New Features* - Merge branch 'akpm' (patches from Andrew), track page size with mmu gather and force flush

Short description of the commit is giving better accuracy when compared to full description of the commit. This suggests that people are being more precise in writing the short description of the commit, whereas in the long description they tend to go out of point and has been more random in nature.