# Grouping and Testing Methods with Clustering Algorithms

Allen Burgett
Normandale College
burgetta@my.normandale.edu

***Abstract*—There are many tools, and in some cases code, available to developers on Android platforms. However, finding the most desirable code and more specifically the most efficient code can be difficult. This paper presents the framework for a technique, EffMethod (Efficient Method), by which a developer could quickly find and incorporate existing efficient code into their project. EffMethod utilizes a modified version of the k-means clustering algorithm to identify similar methods and outputs those methods for testing. After a battery of tests to determine efficiency, the developer is provided with a much smaller amount of methods to choose from.**

***Index Terms*—Program Analysis, Static Analysis, K-Means, Efficient Code, Edit Distance**

## I. Introduction

WHILE application developers have access to large stockpiles of open source code, finding the right code snippets for a project can be time consuming. At times, this could lead to in-house development of code that has already been openly shared. These in-house developments might be riddled with errors and inefficient code, leading to further patching and development, and wasting more time and money. These kinds of setbacks can be costly for any organization, but especially costly to small organizations.

To combat this problem, we gauge the feasibility of comparing open source methods from a variety of applications. Utilizing our EffMethod tool, we will allow the developer to quickly compare several open source applications. The result will give the developer a list of methods and an efficiency rating of those methods. This efficiency will rely on testing for energy and memory usage as well as run-time. This will allow the developer to import pre-made efficient methods into their programs, saving cost and time. Allowing developers to spend more time on the more proprietary aspects of their applications.

To gain access to large repositories of open source android code, we've chosen F-Droid. F-Droid allows us to search for specific types of applications and download the entire program as source code and separate from the APK package found on Google Play. This is particularly helpful as we need examples of a variety of applications as well as access to their source code and .class files.

While this work is currently being tested on the Android platform, we believe that with modification, this technique could be applied to every form of development.

The contributions of this paper are therefore as follows,

- A method for quickly gathering java bytecode on a compiled Android program.
- A technique for identifying similar methods through k-means and edit distance.
- An integrated test suite, that focuses on energy efficiency, memory usage, and runtime.

## II. Background

APIs (Application Program Interface) are used to allow methods to communicate with each other and perform certain pre-defined tasks. API packages contain a library of methods and classes from which a developer would call to perform a certain task within their method. Furthermore, work done on MAPO [10] and SAMOA [6], utilizes API calls along side other properties to determine what those methods do.

MAPO utilizes a two part process for their clustering. First, hierarchical clustering in which the API calls and their sequences are taken from individual methods and are used like letters in a word and grouped according to "family". The results of this are then cross referenced with a similar hierarchical clustering technique, which splits the names of the methods into families as well. The resulting data is a fairly accurate clustering of methods of similar type. Using these ideas, we can extrapolate that methods that make certain specific API calls, especially if those calls are in the same order or similar order, are doing similar things. By clustering those methods that have API call similarity, we can begin to categorize the methods and make predictions as to what they're actually doing.

The final product of the above step in our process will be similar to that of MAPO's. However, where they used a two part hierarchical clustering process, which has a fairly high time complexity, we will attempt to gain similar results with k-means which has a dramtically lower time complexity than MAPO's clustering process. This would allow our process to integrate other testing models to conclude efficiency, without sacrificing speed in the process. The principal question this paper generates is whether this process can actually be accomplished with a non-hierarchical clustering technique, namely k-means.

Based on the research conducted by Pathak et al while developing Eprof [8], we can conclude that some methods are just developed better. Where better is defined as less energy usage. Better can also be defined in terms of user experience. The immediate example of this is, if an application comsumes a dramatic amount of energy in comparison to its competitors, then the user is less likely to use it. A different example of efficiency, which may also effect user experience is, if a method is utilizing a large amount of memory, the user will experience delays in that program and other parts of their of device. In general, users have little patience for slow tasks. While stack memory usage has been analyized for quite some time, there have been many fewer breakthroughs in analyizing heap memory. Specifically sampling heap memory during application execution. Brenschede's [2] work in this area combined with current stack analyzing techniques, will be paramount to analyzing an application or individual method's memory usuage within this project's scope. Therefore, developers have to take energy and memory usage, as well as runtime into account when writing or selecting code.

## III. Technique

The project comprises of two separate parts. First, an effective tool like MAPO [10], will need to be developed, that performs a static analysis of each selected method's API calls [1]. This static analysis will identify the function and similarity of each method in comparison to the others. Our version of MAPO's core research utilizes a variation of k-means explained later in this section, our tool would then categorize each of the selected methods, based on the number of shared API calls and their sequence within the method. The clustering algorithm will make predictions on what the methods do, based on their API calls. This will translate into separate categories that the developer can pick.

To do this applications have to be broken down into methods, with a focus on the calls invoked by the method. Java gives us an easy process by which this can be done: the commandline function 'javap -c'. This breaks down a .class file, which is a compiled .java file, into java bytecode and comments on the java bytecode. By building a script that executes commandline functions, javap can be invoked on an entire application recursively. A simple java program can then parse the comments and extract only the method names and calls. This information is then dumped into a .csv file and provides a summary of the application, its methods, and calls. All of this gathered data, for all applications, needs to be concatendated together so that pre-clustering information can be gathered from it in bulk.

Quailty assurance needs to be performed on the data given. Things like single call methods are excluded, because they're too small to be classified accurately and possibly of no use to the developer (who might find it easier to build the small method themselves, than to search for it). During this quality assurance process, two other important factors must be taken into account. First, if a method has a call that references another part of the program or as we refer to it a "local call", the method being invoked should be broken down into calls and replace the local call with the sequence of calls from the invoked method. This process is handled recursively, with checks in place that the process does not encounter an infinite loop, where a local call is invoked which invokes another local call, which in turn invokes the orignal local call. Once the local calls are replaced with sequences of real API calls, the quailty assurance program then removes any call that is invoked only once across all programs. This is done to insure that those singular calls do not disrupt the mean generation or actual clustering itself.

Our problem presents a unique structural difference to the basic k-means architecture. Traditonally k-means is used with euclidean distance, which cannot be performed on our data set. This distance metric is fundamental to k-mean's inital mean generation as well as its recentering process. To combat this distinction, we utilize a different method to generate our inital means. After we've collected the most appropriate data, we generate means using the calls we've gathered post-quailty assurance. The size of these means are based on the number of overall calls and the amount of means being generated. The number of means being generated varies based on the number of programs that are incorporated into the experiment. Within our version of k-means, our randomly generated means utilize edit distance to calculate their distance from the imported data. Our version of edit distance is based on how many removes and adds are required to make our random mean the same as the original method. Where the calls act as characters within a string and certain edits are required to make the strings the same, as shown in Figure 1.

Our current configuration of k-means has foundational similarities to k-means as well as some drastically different modifications. As it exists, our k-means takes in the randomly generated means as well as the full real-data set. During the first pass, each real-method is assigned to a mean based on its edit distance from each mean. If the lowest distance is split between more than one mean, a mean is randomly picked from that set and the method is assigned to it. Once all methods have been assigned to a cluster, the clusters are submitted to a recentering process. If any clusters are empty coming into this recentering process, then a poll is done within all of the clusters and the method with the largest overall distance from its mean is moved to the empty cluster. Next, each method inside of a cluster is polled regarding which call is contained within each index. This polling data is compared against the other methods' polling data and the call with a plurality of votes within each index is assigned as the new index for the mean. Any remaining empty indexes of the mean are filled with random calls from the post-quality assurance call set.

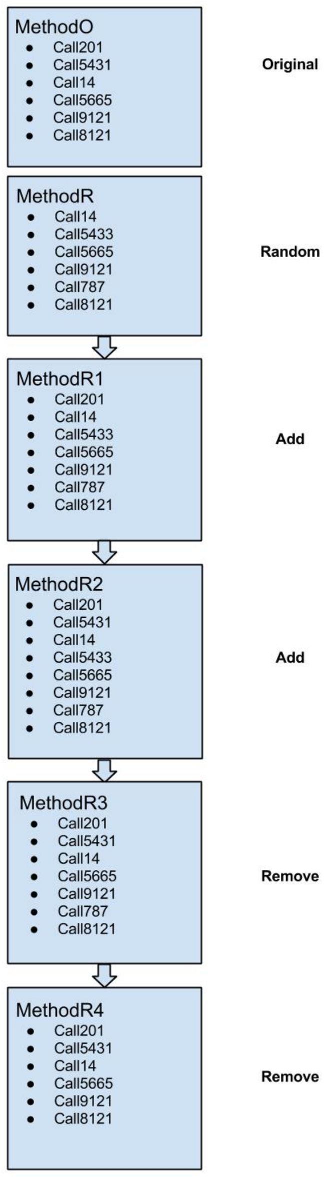

Figure 1. Our version of edit distance

The process starts again and all of the methods are compared to the new means and assigned as they were on the first iteration. The recentering process workes the same each iteration. As of now, iterations are chosen arbitrarily in an attempt to find the optimum amount.

The second part of this process identifies the most efficient methods out of those that have been clustered. Once the clustering is complete and the developer has decided which category of method to compare, a battery of tests will be executed based on: energy usage, memory usage, and method run-time. For energy usage, we will utilize a tool called Eprof, developed by Purdue and now licensed by a company called Microenergetics. Eprof breaks down applications' energy consumption by each method [8]. We will be testing memory usage by a combination of stack and heap sampling BHeapSampler [2], developed by Brenschede. Using an assortment of benchmarking tools or refactoring, we will compare the run-time of the individual methods. After testing is completed, the outputs of each method will be sorted based on probability distribution. This will give us a percentage based in comparison to the other methods. An overall rating will be assigned as an average of the method's percentages in each category. Results will be displayed to the user, divided into the four categories of overall, energy usuage, memory usage, and runtime. Those categories will be sorted most efficient to least, so that the developer can make an educated choice about the best method for their project.

## IV. Preliminary Results

Our current configuration of k-means has provided mixed results. Some methods have shown similarity within their respective cluster, while others within the same cluser have no apparent similarity with any other method. Further yet, at least 10% of these clusters have absolutely no similarity amongst member methods.

The more interesting analysis so far is, that iterations over 100 seem to have no possitive affect on overall clustering; with optimal iterations seeming to be between 40 and 80. In fact, iterations over 100 have a 50% chance of converging all of the methods into one singular cluster.

## V. Discussion of Results

Some inital explanations of our current results, have to do with how the data is being generated, refined, and processed within k-means. The smallest, but still important, flaw is the original capturing of data itself. When taking in method names, the parsing program does not strip their arguements as well. After reviewing the original data, we can see that about .5% of the original methods have overloaded counterparts. These are not distinguishable from one another and are therefore labeled as the same method, which correlates to at least .5% of our inital data being useless, but still used. The simpile soultion to this problem is to strip the arguements with the method names, which insures overloaded methods would not be rolled together.

The second problem originates from our current disregard for local calls. Where we define local calls as calls that are made to other parts of the program. This problem derives itself from the complicated task of recursively replacing local calls with a list of the external API calls of the invoked internal method. As our model currently stands, local calls are removed during the quality assurance process and disregarded overall. This ensures that the local call, which is unlikely to be found in methods from other programs, does not distort the overall data that is to be clustered. Longterm development will have to include a process for including the external calls of a local call. This process will also give us a better picture of the method and what it does.

The third problem comes from the randomness of our method placement during the cluster assginment phase of our k-means process. It is possible, with our current configuration, that methods with similar sequences of calls would be randomly placed into separate clusters. Then if they're not polled out of those clusters, they will never converge into a singular cluster. A better process needs to be designed in order to prevent this eventuality during the cluster assignment phase.

The final conceived problem with our current configuration is with the recentering process as a whole. As of now, clusters rely heavily upon the polling data collected from the methods. However, the current polling does not account for differences in size or locations of similar sequences. Two methods may be similar, for example a method with a size of nine might contain the same sequence as a method with a size of four. The method with a size of nine might invoke some common call five or six times before it invokes the similar sequence. While these two methods have commonality, they are unlikely to be clustered together, except by accident, because they would poll to change different indexes of their respective mean. A process needs to be developed to handle polling for sequences as a whole and not individual indexes.

## VI. Conclusion

Mobile applications are often run in low resource environments, making energy and memory huge concerns for both developers and users. These concerns, including run-time, are aspects of applications that will affect users and be the basis of their application download choices. This means that finding code that operates efficiently is a huge concern for developers. With EffMethod, we have laid out a framework in which a developer could easily identify and implement efficient code. This tool will save both time and money for organizations and individual programmers alike. We have also laid out an arguement for using k-means as a time-conscious alternative to MAPO's hierarchical approach to the initial clustering of this problem.

## VII. Further Considerations

The randomness of our mean generation presents its own unique problem: finding a comparable example of each unique type of method within the given set of applications. This inadequacy has led to the development of a possible alternative to API call sequence clustering. By classifying methods by the packages it invokes instead of the sequence of calls, one could cluster those methods that have an overall similarity. Many deviations of this could be tested and we offer this as an alternative branch to our current process.

## VIII. Future Work

Our long term development goals are as follows:

- Implement the energy and memory usage and run-time sorting process.
- Release EffMethod as an Eclipse plug-in.
- Generalize the tool to work on other languages. Currently this is being developed for Android, however it is not a strech to assume it will also work for standard Java programs. Which also points to the possibility of the overall idea being applied to other languages outside of the Java umbrella.
- Build-in a measure that will track user changes to code. This could be used to develope a genetic algorithm similar to GenProg [4] which has been used to patch legacy code. A modified version of GenProg could potentially modify the open source code selected by the developer, so that it fits easily within their current project.
- Build-in a measure that informs the developer of updates to the code they selected for their project.
- Add EffMethod to other major IDEs, like Visual Studio.

## IX. Related Work

MAPO, developed by Zhong et al, clusters methods based on their API calls, order of calls, and to some extent method name [10]. SAMOA, developed by Minelli et al, analyzes applications by development history, API calls, and source code [6]. The work done by Barstad et al. runs a static analysis on student's code to determine whether its "good code" or "bad code" [1]. Their tool was better at identifying "good code" than "bad code". Pathak et al developed the tool Eprof to measure energy usage of methods contained within applications [8]. Brenschede developed a tool called BHeapSampler, to analyze heap data in a java application [2]. Murphy et al dives deeper into refactoring, which could aid in our run-time tests [7]. The work provided by Hecht et al, resulted in an effective tool called Paprika, that can detect antipatterns in Android applications [3]. Similarly, Verloop et al, measured code smell in four Android applications [9]. Le Goues et al utilized genetic algorithms to patch legacy code with a 94% success rate [4]. Machiry et al developed a tool called Dynodroid, that generated input sequences and found bugs in the input sequences of 5 of the top 1000 Android applications [5].

## Acknowledgment

The author would like to thank the University of Colorado at Colorado Springs and the National Science Foundation's Research Experiences for Undergraduates program, who's Grant 1359275 funded this research.

## References

[1] V. Barstad, M. Goodwin, and T. Gjøsæter. Predicting source code quality with static analysis and machine learning. *Norsk Informatikkonferanse (NIK)*, 2014.

[2] A. Brenschede. Graph-based performance and heap memory analysis. In *Proceedings of the 15th Java Forum Stuttgart*, 2012.

[3] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. *Detecting Antipatterns in Android Apps*. PhD thesis, INRIA Lille, 2015.

[4] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[5] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[6] R. Minelli and M. Lanza. Software analytics for mobile applications–insights amp; lessons learned. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 144–153, March 2013.

[7] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *Software, IEEE*, 25(5):38–44, 2008.

[8] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42. ACM, 2012.

[9] D. Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013.

[10] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.