The Uses of Machine Learning Technique to Improve Software Quality and Testing

Amy Aumpansub
College of Computing and Digital Media
DePaul University
Chicago, USA
amy.aump@gmail.com

Jenney Chang
College of Computing and Digital Media
DePaul University
Chicago, USA
jdzchang@gmail.com

Darrell Lanel
College of Computing and Digital Media
DePaul University
Chicago, USA
darrelllane25@gmail.com

***Abstract*— During the past few years, several machine learning techniques have been utilized to increase automated Software Quality and Testing and was found to be more effective than traditional methods. To assess and improve software quality, this study developed an approach which combined a machine learning technique called K-Means Clustering and the well-known CK-Metrics to assess and predict software quality and cluster all classes based on their quality.**

Keywords—CK Metrics, K-Means Clustering, Software Quality and Testing

I. Introduction

Our study focuses on quality analysis and optimization of software for Android Image loaders: Photo View, Picasso, and Android Universal Image Loader. A three-step method was utilized: Software quality assessment using CK-Metrics, Quality Prediction and Clustering using K-Means (Machine Learning), and refactoring and testing software predicted as low-quality. The CK-Metric values were used for quality assessment and trained as inputs of the K-Means Machine Learning model. The machine learning model calculated the quality of classes and grouped them into 3 clusters (low, medium, high-quality). The low-quality classes of Photo View, Picasso, and Android Universal Image Loader account for 6.25%, 5.47%, 10.65% of the total number of classes. The classes grouped as "low-quality" allowed for quick targeted refactoring and testing. The final output of the study is an improved version of the Image Loaders with higher quality.

II. Problem Statement

CK-Metrics have been widely used to assess the overall quality and evaluate attributes of object-oriented software development such as classes, inheritance, and encapsulation [1]. Traditional methods to improve software quality may be time-consuming if software developers need to detect faults in many classes.

To address this problem, the study utilized both CK-Metrics and machine learning to develop a strong predictive model to examine the quality of software at class level and optimize the improvement process. The K-means clustering model groups all classes of each software based on CK-Metrics which enables software engineers to focus on low-quality classes during the improvement process. This approach increases speed and effectiveness of the quality improvement.

III. Methodology

The three-step method of quality analysis and optimization outputs a higher quality version of the Image Loaders. The details are discussed below:

A. *Software Quality Assessment*

The quality of each Image Loader was assessed using CK-Metrics, containing six metrics: Weighted Method per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response for Class (RFC), and Lack of Cohesion (LCOM). The metric values of each Image Loader were extracted from the source code using the Understand Tool; and compared across image loaders to evaluate the degree of object-oriented software implementation. The metric values of each image loader were trained as inputs for the Machine Learning model in the next step. A complete analysis of CK metrics was discussed in section IV (Results).

B. *Quality Prediction and Clustering*

The quality of each Image Loader at class level was predicted and clustered using the K-means clustering model. K-means clustering model is an unsupervised machine learning technique designed to group similar data points together and discover underlying patterns shared among data groups [2]. This algorithm was applied to group classes of each Image Loader into three clusters: low-, medium-, and high-quality based on the CK-Metrics extracted in Step 1. The NumPy, Pandas, and Scikit-learn package in "Anaconda" tool was utilized to preprocess and train data, build models, and predict the quality of each class. The seaborn and Matplotlib were used for visualizing results.

1) *Training Dataset*

There are three datasets for each Android Image loader: Photo View, Picasso, and Android Universal Image. The total number of observations in the dataset is equivalent to the number of classes in the source code. Each dataset contains six independent numeric attributes including WMC, DIT, NOC, CBO, RFC, and LCOM. There is no dependent variable as this study focused on K-Mean Clustering, which is an unsupervised classification with no target label.

2) *Data Exploration*

Pearson correlation was utilized to evaluate the strength of relationship between two metrics [3]. The coefficient values were examined to identify the relationship between two metrics ranges from -1 to 1. A correlation coefficient of 1 implies a positive and strong linear association, whereas a coefficient of -1 implies a negative and strong linear association. A correlation coefficient of 0 means two metrics are independent and not related to each other. The correlation analysis was discussed in Section IV: Results.

3) Data Preprocessing

a) Z-score Normalization

K-mean clustering algorithm groups the data by computing the distance between them and is sensitive to outliers and range of data values. Thus, the raw data needs to be normalized before fitting the K-mean clustering model. The Z-score normalization was applied to raw values to handle the outlier and standardize each feature (metric) by subtracting the mean and then scaling to unit variance. The normalized data have a normal distribution with a mean of 0 and a standard deviation of 1.

b) Dimensionality Reduction

The Principal Component Analysis (PCA) is a method for reducing the dimensionality of data sets by transforming a set of independent attributes into a smaller set of attributes while still preserving most of the information in the original set. The PCA can be used before K-mean clustering to visualize high dimensional data and reduce computation cost for building K-mean models. This study performed the PCA techniques to reduce dimension of inputs from six- to two-dimensional vectors, containing principal component 1 and 2.

4) K-Mean Clustering Model

After transforming the raw data, the K-means clustering models were trained separately for each Image Loader using the datasets containing two features (principal components) to do the unsupervised classification. The K-mean models were built with the goal to partition all classes of each Image Loader into three clusters.

The models were initialized with a maximum number of 300 iterations, three clusters, and a relative tolerance of 0.0001 with a classical EM-style algorithm. Each model was trained by initializing the cluster centroids for each cluster and computing the distance of each observation from the centroids. The model randomly assigned points to each cluster and iterated the process until the centroids were unchanged. The models of PhotoView, Picasso, and Android Universal Image Loader took 2, 6, and 11 iterations. The final value of centroids for each cluster from the last iteration are shown in Table I.

TABLE I. CLUSTER CENTROIDS OF EACH IMAGE LOADER

Image Loader	**Cluster Centroid**		
	Low	***Medium***	***High***
PhotoView	[6.27, 0.05]	[-0.66, 2.78]	[-0.33, -0.27]
Picasso	[3.73, -0.16]	[-0.60, 1.06]	[-0.34, -0.69]
Android UIL	[4.55, -0.65]	[-0.22, 1.49]	[-0.61, -0.60]

Each K-means clustering model contains three cluster centroids as shown on Table 1, which are related to the quality of the classes. We used the K-means model to predict the cluster number for each observation based on its distance to the nearest mean or cluster centroid. Each class was assigned to one cluster based on their quality extracted from CK metrics. The low-quality classes were further examined for Anti-patterns, refactoring, and testing in the next step.

C. Software Quality Improvement

The low-quality classes predicted from the K-means model were targeted for refactoring and testing. The low-quality classes were analyzed in-depth to reveal structural and implementation problems in each image loader. The low-quality classes were refactored, and unit tests were created to ensure performance of each program remained unchanged. Combining the machine learning approach with a deep understanding of the system enabled us to accurately detect system flaws and improve its quality during refactoring and testing.

a) Anti-pattern Detection

The code of low-quality classes was examined to detect design, architecture, and implementation flaws. UML, sequence, and use-case diagrams were used to examine the logical flow of user interaction with the system. Diagramming provided a way to detect system flaws and improve its quality during refactoring and testing.

For example, the “PhotoviewAttacher” class from Photo View Image Loader was predicted as a low-quality class and had several design smells such as lazy class and contains unrelated methods to its main functionality. The goal during refactoring process was to fix these design smells.

b) Refactoring

The refactoring process was implemented in Android Studio, due to the image loaders being developed for the Android Operating System. Android Studio provides modules and libraries to build and run each image loader. All dependencies of Image Loaders were specified in the build.gradle file.

The example sets of refactoring operations and Testing performed on each Image Loader are as follows:

1) Photo View Image Loader

The first Anti-pattern detected in this Image Loader is that the Class Util, GestureCustomDetectors, Photoview, and PhotoViewAttacher contain duplicated codes and dead codes, so these classes were refactored by removing duplicated codes and extract new functions as follows:

- RemoveDuplicatedCodes(From Class PhotoView and Class PhotoViewAttacher)
- RemoveDuplicatedCodes(From Class Util and Class GestureCustomDetectors)
- ExtractNewField (*mGestureDetector*, From Class CustomGestureDetector)

- ExtractNewFunction(*DetectExistingFocus(*mGesture*Detector)*, From Class GestureCustomDetectors)
- ExtractNewFunction(*UpdateFocus (*mGesture*Detector)*, Class GestureCustomDetectors)

The class "PhotoviewAttacher" and the class "GestureCustomDetectors" are large god or blob classes that have numerous responsibilities, while other classes such as Photoview holds only data. This implies another code smell because Class "Photoview" is large, having many functionalities and many attributes.

To fix the detected design antipattern, the class "GestureCustomDetectors" and "PhotoviewAttacher" can be refactored by extracting it into subclasses to distribute responsibilities to its child classes which implement functionalities related to its class. The refactoring operations include: ExtractSubClass, MoveFields, MoveMethods, Pull Up/Down Fields/Methods, RenameFields, RenameMethods, RemovedSettingMethods, and RemoveDuplicatedCode.

- ExtractClass (Focus, From GestureCustomDetectors)
- PullDownFields (*lastFocusX, lastFocusY*, From Class GestureCustomDetectors, Class Focus)
- PullDownMethod (*DetectFocus(mDetector)*, From Class GestureCustomDetectors, Class Focus)
- PullDownMethod (*updateFocus(mDetector)*, From Class GestureCustomDetectors, Class Focus)
- ExtractClass(Scaler, From PhotoviewAttacher)
- MoveFields (*max_scale, min_scale, mid_scale*, From Class PhotoviewAttacher, To Class Scaler)
- MoveMethods (*getMinimumScale (), getMinimumScale (), getMinimumScale (),* From Class PhotoviewAttacher, To Class Scaler)
- MoveMethods(*setMidiumScale (), gsetMaximumScale (), setMinimumScale ()*, From PhotoviewAttacher, To Scaler)
- RemovedSettingMethods(*setDefaultMaxScale (), setDefaultMinScale()*, From Scaler)
- Renamefields and Methods to increase clarity
- ExtractClass(Edges, From PhotoviewAttacher)
- Movefields (*all horizontal edges, all vertical edges*, From PhotoviewAttacher, To Edges)
- RemovedSettingMethods(*setDefaultVerticalEdges (), setDefaultHorizontalEdges()*, From Class Scaler)
- Rename fields and Methods to increase clarity

2) *Picasso Image Loader*

The classes marked as low-quality were Picasso, Dispatcher, Request Creator, and Bitmap Hunter. All the low-quality classes had high WMC and LCOM values, indicating classes with high complexity, little cohesion, and generally bad design. Further analysis of classes revealed:

- Architecture smell: God Component (Picasso)
- Design smell: hub-like modularization (Picasso) and insufficient modularization (Picasso, Dispatcher, and Bitmap Hunter)
- Implementation smells for all classes: long parameter list, long statements, complex method, complex conditional, dead code, and magic numbers

The dependency inversion principle was attempted to remove the hub-like or insufficient modularization and god class features from Picasso. Creating a dependency on an abstraction increases encapsulation between classes. Classes were also pulled out of Picasso to increase single responsibility.

Dispatcher, Bitmap Hunter, and Request Creator were refactored to remove implementation issues such as magic numbers, dead code, and redundancies. The specific changes are listed below:

- Extract Class (Listener, From Picasso)
- Extract Class (RequestTransformer, From Picasso)
- Extract Class (CleanupThread, From Picasso)
- Move Fields (targetToAction, targetToDeferredRequestCreator, referenceQueue, From Picasso)
- Removed Magic Numbers (90, 180, 270, From getExifRotation)
- Removed Magic Numbers (90, 270, From transformResult)
- Move Fields (hunterMap, failedActions, pausedTags, batch, From Dispatcher)
- Removed Magic Numbers (21, 16, From getPlaceHolderDrawable)

3) *Andriod Univeral Image Loader*

Several Anti-patterns were found in the class LimitedMemoryCache, ImageLoader, LimitedAgeDiskCache, BaseImageDownloader, PauseOnScrollListener, and DisplayImageOptions. The classes contained duplicated code, long methods, and long parameter lists; refactoring of these classes consisted of removing duplicated codes, minimizing long methods, and long parameter lists.

- RemovedDuplicatedCode (From Class LimitedMeoryCache)
- RemovedDuplicatedCode (From Class LimitedAgeDiskCashe)
- ReworkedLongMethod (From Class BaseImageDownloader and PauseOnScrollListner)
- ExtractedNewFunctions(ImageSize(mDisplayImage), Class (ImageLoader)
- MoveFields(imageResOnloading, imageResForEmptyURL, ImageResOnFail, from Class DisplayImageOptions, To Class intImages)
- MoveFields (imageForEmptyURi,, imageOnLoading, imageForEmptyURI, imageOnFail, From Class DisplayImageOptions To Class ImageDrawables)
- MoveFields (preProcessor, postprocessor, and displayer, From Class DisplayImageOptions, To Class BitmapProcessing)
- MoveFields(resetViewBeforeLoading, cacheInMemory, and cacheOnDisk, from Class DisplayImageOPtions to Class validateOptions.)

The second anti-pattern detected was a switch statement in the Class BaseImageDownloader. This was class showed to have a cluster of methods and a switch statement that calls these methods during certain cases, leaving dead code within the class.

Refactoring Class BaseImageDownloader consisted of extracting the switch case statements. Each switch case statement was put into their own class; each class was extended from their parent class with all test passing.

- ExtractedClass(getStreamFromNetwork, from BaseImageDownloader)
- ExtractedClass(getStreamFromFile, from BaseImageDownloader)
- ExtractedClass(getStreamFromContent, from BaseImageDownloader)
- ExtractedClass(getStreamFromAssets, from BaseImageDownloader)
- ExtractedClass(getStreamFromDrawable, from BaseImageDownloader)
- ExtractedClass(getStreamFromOtherSources, from BaseImageDownloader)

c) Testing:

To ensure the functionality remains unchanged after refactoring, several JUnit 4.0 test cases were implemented on the refactored classes and methods. The test cases were created for refactored classes. To use the local JUnit test for Android, its dependencies were specified on build.gradle file.

All tests were run as test suit with JUnit Runner with these annotations:

@ RunWith(Suite.class)

@ Suite.SuiteClasses({TestClasses}).

IV. Results

A. Quality Analysis Comparison

The CK metrics extracted from Picasso, Photo View, and Android Universal Image Loader were compared for assessing their quality. The comparison graphs are shown on Fig. 1, and the quality analysis from six metrics are as follows:

1) Weight Method per Class (WMC): Measures the complexity of classes among three image loaders. Picasso was found to have a higher complexity value than Android UIL and Photo View. The value obtained for Picasso is 11.78; a moderate complexity value. Android UIL was found to have a moderate complexity value of 11.11. A low complexity value of 9.58 was obtained for Photo View (Fig. 1). Lower values of WMC are favorable than higher values.

2) Depth of Inheritance Tree (DIT): Measures the max distance of a given tree node to its root, a moderate value of 1.58 was obtained for the Picasso Image Loader. The image loader with the lowest value is the Android UIL, at value of 1.48 (Fig. 1). A high value of 1.68 was obtained for the Photo View. A lower value for the DIT metric is favorable; lower

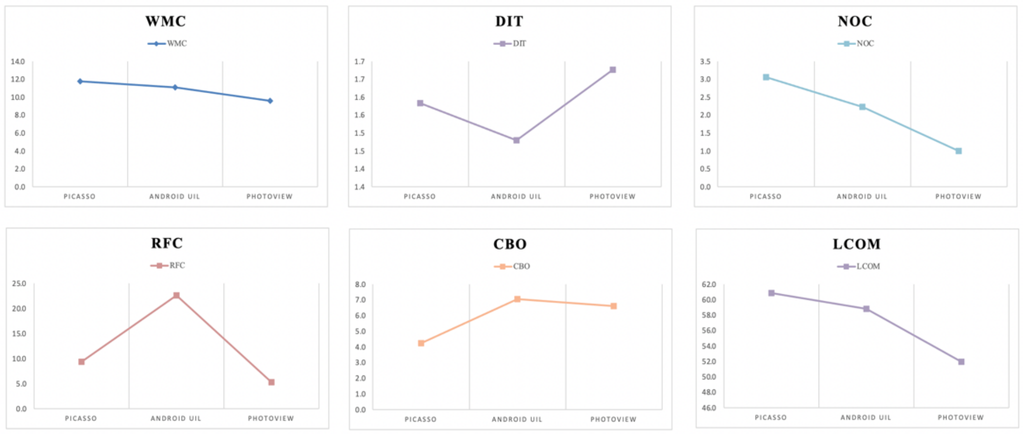


Fig. 1. Comparison Graphs of CK Metrics

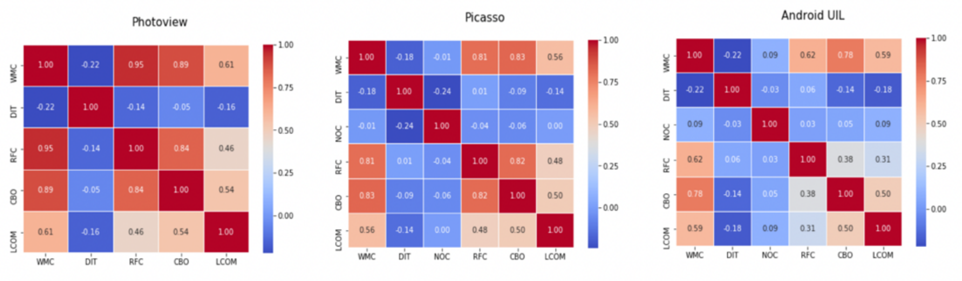


Fig. 2. Correlation Coefficient Values of 3 Image Loaders

values indicate higher potential for reuse and manageable behavior complexity.

3) Number of Children (NOC): Measures the number of direct children per class. Picasso Image loader was found to have a high value of 3.06. A moderate value of 2.23 was obtained for the Android UIL; which was significantly lower than the Picasso. Photo View had the lowest value of 1.00 (Fig. 1). Low values of NOC are an indication of good design patterns within programs; higher values tend to indicate unfavorable subclass inheriting design patterns from a parent class.

4) Response for Class (RFC): Measures multiple method invocation by a single response. Picasso had a moderate value of 9.34. A high value of 22.64 was obtained for Android UIL; a significantly higher value compared to Picasso and Photo View. Photo View had the lowest value at 5.32 (Fig. 1). Low values of RFC are preferred; and are tend to indicate good manageability and high levels of polymorphism.

5) Coupling Between Object Classes (CBO): Objects with tight coupling change together, increasing complexity and decreasing the manageability of a program. The Picasso Image Loader had a low value of 4.25; significantly lower than the values of Android UIL and Photo View. A high value of 7.05 was obtained for Android UIL; a moderate-to-high value of 6.61 was obtained for Photo View (Fig. 1). Low values in CBO are preferred and are good indicators that classes are potentially reusable.

6) Lack of Cohesion Metric (LCOM): Cohesiveness measured in the LCOM method for the Picasso Image Loader returned a high value of 60.86; and Android UIL had a moderate to high value of 58.85. The Photo View Image Loader had a lower value of 52.00. High LCOM values indicate a greater level of complexity and a lower level of cohesion.

Overall, Photo View has the highest quality as it has the lowest average values of all metrics except DIT. Picasso seems to have the lowest quality as it has the highest average values of WMC, NOC, and LCOM, whereas Android Image Loader has the highest average values of RFC and CBO.

B. *Correlation Coefficient Analysis*

Analyzes quality across image loaders, the coefficient values of six metrics were computed as shown on Fig. 2. These values explain the strength and direction of relationship between two metrics ranging from -1 to 1.

a) Photo View: The WMC metric has a very strong positive linear relationship with RFC and CBO. A correlation coefficient greater than 0.85, shows as the CBO values increase, the RFC and CBO values also increase. A moderate positive association between CBO and LCOM metrics (r = 0.54), while the DIT metric has a weak negative association with the other five metrics.

b) Picasso: The CBO metric has a strong positive linear relationship with WMC and RFC. A correlation coefficient greater than 0.8, indicates that as the CBO increases, the WMC and RFC values also increase. A moderate positive association between WMC and LCOM metrics (r = 0.56). A weak negative association between WMC and DIT metrics (r = - 0.18).

c) Android UIL: There is a strong positive association between WMC and CBO metrics (r = 0.78), as the CBO rises, the WMC values also rise. The WMC has a moderately positive relationship with RFC and LCOM metrics. The DIT metric has a weak negative association with WMC, CBO and LCOM with r values of -0.22, -0.14, and -0.18, respectively. The NOC is not correlated with RFC and CBO as its coefficient is close to 0.

C. *Predicted Quality and Cluster*

The K-Means model was used to predict the quality of classes based on the values of six metrics; classes were grouped into three clusters: low-, medium-, and high-quality.

- Fig 3 shows the three quality clusters classes in each image loader predicted by K-means models plotted with the axis values of principal components.
- Principal components extracted from six CK-Metrics
- Classes assigned to a low-quality cluster tend to have a higher value of both principal components.
- Classes assigned to a high-quality cluster have a lower value of both principal components.
- Among three image loaders, Android UIL loader has the most low-quality classes or about 10.6% of total classes. Photo view and Picasso has smaller proportion of low-quality classes, about 6.25% and 5.47% of total classes.
- All three programs have majority of data points in high-quality clusters, implying the good fit for the K-Means clustering model.

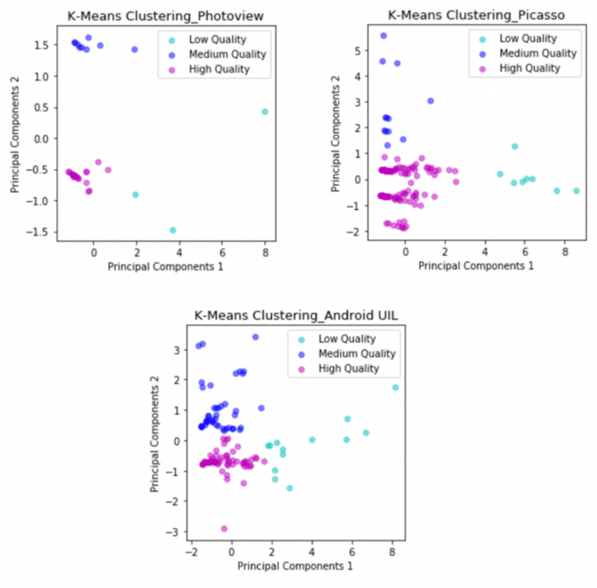


Fig. 3. Predicted Quality Cluster of Image Loaders

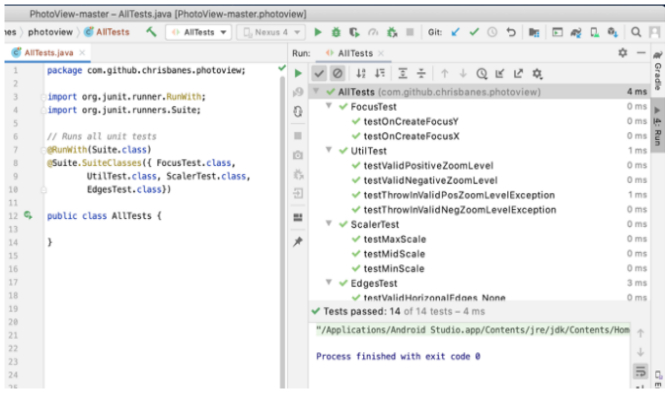
D. *Quality Improvement*

The refactoring and testing were performed on a low-quality cluster predicted by The K-means models, labeled in light-blue color in Fig. 3. The most common anti-patterns found in this study were Blob class, Lazy class, Wide Hierarchy, and long methods. The study shows the metric values of refractored classes are lower than classes in original source codes. For testing, the JUnit test suite of each Image Loader were run before and after refactoring, and the results showed that the refactored programs passed all tests, indicating the maintainability of codes after improving the quality of codes.

E. *JUnit Testing Results*

To ensure the functionality has remained unchanged after refactoring, several JUnit 4.0 test cases were implemented on the refactored classes and methods. The test cases were created for refactored classes and all refactored programs passed all tests which confirmed the correct operations of refactored classes. The test results were shown below.

1) *Photo View*

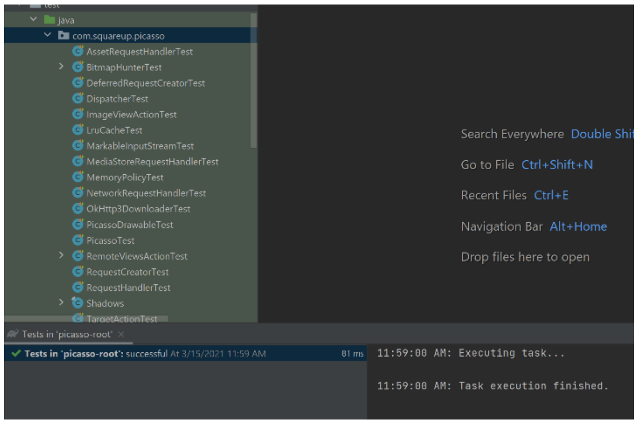


```
package com.github.chrisbanes.photoview;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

// Runs all unit tests
@RunWith(Suite.class)
@Suite.SuiteClasses({ FocusTest.class,
        UtilTest.class, ScalerTest.class,
        EdgesTest.class})

public class AllTests {

}
```

```
AllTests (com.github.chrisbanes.photoview)          4 ms
  FocusTest                                          0 ms
    testOnCreateFocusY                               0 ms
    testOnCreateFocusX                               0 ms
  UtilTest                                           1 ms
    testValidPositiveZoomLevel                       0 ms
    testValidNegativeZoomLevel                       0 ms
    testThrowInValidPosZoomLevelException            1 ms
    testThrowInValidNegZoomLevelException            0 ms
  ScalerTest                                         0 ms
    testMaxScale                                     0 ms
    testMidScale                                     0 ms
    testMinScale                                     0 ms
  EdgesTest                                          3 ms
    testValidHorizonalEdges_None                     0 ms
Tests passed: 14 of 14 tests – 4 ms
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Hom

Process finished with exit code 0
```

Fig. 4. PhotoView JUnitTest on Android Studio

2) *Picasso Image Loader*

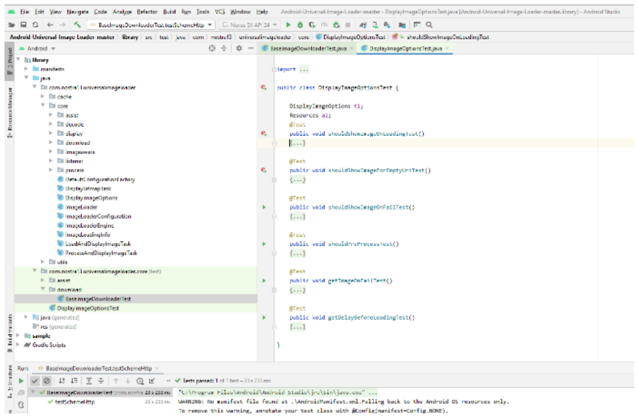


```
java
  com.squareup.picasso
    AssetRequestHandlerTest
    BitmapHunterTest
    DeferredRequestCreatorTest
    DispatcherTest
    ImageViewActionTest
    LruCacheTest
    MarkableInputStreamTest
    MediaStoreRequestHandlerTest
    MemoryPolicyTest
    NetworkRequestHandlerTest
    OkHttp3DownloaderTest
    PicassoDrawableTest
    PicassoTest
    RemoteViewsActionTest
    RequestCreatorTest
    RequestHandlerTest
    Shadows
    TargetActionTest
```

```
Tests in 'picasso-root': successful At 3/15/2021 11:59 AM    81 ms
11:59:08 AM: Executing task...

11:59:08 AM: Task execution finished.
```

Fig. 5. Picasso JUnitTest on Android Studio

3) *Android Universal Image Loader*



```
import ...

public class DisplayImageOptionsTest {

    DisplayImageOptions ti;
    Resources al;
    @Test
    public void shouldShowImageOnLoadingTest()
    {...}

    @Test
    public void shouldShowImageForEmptyUriTest()
    {...}

    @Test
    public void shouldShowImageOnFailTest()
    {...}

    @Test
    public void shouldPreProcessTest()
    {...}

    @Test
    public void getImageOnFailTest()
    {...}

    @Test
    public void getDelayBeforeLoadingTest()
    {...}

}
```

```
BaseImageDownloaderTest.testSchemeHttp
Tests passed: 1 of 1 test – 23 s 233 ms
BaseImageDownloaderTest   23 s 233 ms
  testSchemeHttp          23 s 233 ms
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
WARNING: No manifest file found at .\AndroidManifest.xml.Falling back to the Android OS resources only.
To remove this warning, annotate your test class with @Config(manifest=Config.NONE).
```

Fig. 6. Android JUnitTest on Android Studio

V. Conclusion

Our study utilized CK-Metrics and machine learning to improve software quality. The K-mean clustering models were trained, used to predict quality and group all classes of each software based on measures provided by CK-Metrics. This approach enabled us to focus on low-quality classes during the refactoring process, which increased speed and effectiveness of software quality improvement. Among three image loaders, Photo View has the highest quality based on its CK metrics, and has the small proposition of low-quality classes predicted by the K-Mean Clustering model. On the other hand, Picasso tends to have the lowest quality as it has the highest average values of WMC, NOC, and LCOM, whereas Android Image Loader tends to have the lowest quality as it has the highest average values of CK metrics and has the largest proposition of low-quality classes. The refactoring and testing were performed on the low-quality classes in which their quality was improved as the codes have lesser design smell and their value of CK-metrics are significantly lower.

a) *Future Studies*

The source codes of other open-source software can be used to extract metrics and train the K-means models to ensure the method implemented in this study can be effectively applied to other types of software with no limitation. Refactoring and testing can be further implemented on the medium-quality classes to maximize the quality of the image loader.

b) *Team Member Resposibilities*

Our team divided tasks equally. Each team member was assigned to one image loader to detect code smells, implement refactoring and testing, and write reports for his/her image loader. The details of tasks done by each member are as follows:

Amy Aumpansub

- Photo View Image Loader
  a) Extracted CK Metrics from source codes
  b) Detected anti-patterns
  c) Refactored and tested classes
  d) Writing the report for Photo View
- Built K-mean clustering models
- Assisted in writing report

Jenney Chang

- Picasso Image Loader
  a) Extracted CK Metrics from source codes
  b) Detected anti-patterns
  c) Refactored and tested classes
  d) Writing the report for Picasso
- Provided the open-source codes for the team
- Assisted in writing and proofreading the report

Darrell Lane

- Android Universal Image Loader
  a) Extracted CK Metrics from source codes
  b) Detected anti-patterns
  c) Refactored and tested classes
  d) Writing the report for Android UIL
- Analyzed CK-metrics across 3 image loaders
- Assisted in writing report

References

[1] Y. Suresh, L. Kumar, S. Ku. Rath, "Statistical and Machine Learning Methods for Software Fault Prediction Using CK Metric Suite: A Comparative Analysis", International Scholarly Research Notices, vol. 2014, Article ID 251083, 15 pages, 2014.

[2] K. Nakamichi, K. Ohashi, I. Namba, R. Yamamoto, M. Aoyama, L. Joeckel, J. Siebert, J. Heidrich, "Requirements-Driven Method to Determine Quality Characteristics and Measurements for Machine Learning Software and Its Evaluation", IEEE 28th International, pp. 260-270, 2020

[3] J. Ian and C. Jorge "Principal component analysis: a review and recent developments", The Royal Society Publishing, April 2016