IMA Project 1: God Classes

Navdeep Singh Bedi

April 26, 2024

God class is the classe in OOP which violate the single responsibility principle i.e, it provide too many functionalities inside the same class and thus make the class higly unfocused.

The main goal of the project is to use clustering to to support automated refactoring of God classes.

The project goes as follows:

1. We start with identifying God classes inside the given *xerces2-j-src* repository.
2. Next we extract the feature vectors of the classes based on the class methods and attributes and field methods and attributes.
3. Then we clustering algorithms such as k-Means and Agglomerative to cluster the God classes based on the feature vectors.
4. Finally we measure the quality of the God class partitions using Precision and Recall scores.

0 Code Repository

The code and result files, part of this submission, can be found at:

Repo: https://github.com/infoMA2023/project-01-god-classes-navdeeps350.git
Commit: Commit ID of your submission.

1 Data Pre-Processing

The data preprocessing takes place in two parts:

1. Identification of God classes.
2. Extraction of feature vectors from the God classes.

The steps for the preprocessing are given below.

Class Name	# Methods
XSDHandler	118
DTDGrammar	101
XIncludeHandler	116
CoreDocumentImpl	125

Table 1: Identified God Classes

1.1 God Classes

To identify the God classes we parse the source code in given repository using *javalang* library of Python and count the number of methods in each class without counting the methods in the subclass of the class. This gives the number of methods present in each class of the repository, and then with the help of the given eq.1:

$$God(C) \iff |M(C)| > \mu(M) + 6\sigma(M) \tag{1}$$

where M(C) is the set of methods in class C; M is the set of all methods across all classes The god classes I identified, and their corresponding number of methods can be found in Table 1.

The code for this step can be found in *find_god_classes.py* file. To run the code, the user needs to provide the path (*resources/xerces2-j-src*, in this case) to the repository as an argument to the script. The output of the script is a file named *god_classes.csv* in the *results_csv* folder.

1.2 Feature Vectors

In the second step, we extract the feature vectors of the extracted God classes in the previous step. This is done as follows:

1. First, we select only those class matching the input file name, to skip inner classes.

2. Then for this class in each of the file extracted, we find the field and the method names present in the class and also the fields and methods accessed in the particular method in that class. Doing this we only take the methods and fields present in the God class.

3. We populate the nested dictionary with the method name as a key and the corresponding valid field and method names present in that method with their counter (inner dictionary) as the value.

4. Then we create aq dataframe with all the methods present in the class as rows and all the valid fields and methods in class as columns.

5. Finally we populate the data frame accordingly with the help of dictionary and replace the counter value with 1's and 0's if the corresponding valid field and method is accessed in the given method or not.

Class Name	# Feature Vectors	# Attributes*
XSDHandler	106	194
DTDGrammar	91	102
XIncludeHandler	108	183
CoreDocumentImpl	117	66

Table 2: Feature vector summary (*= used at least once)

Table 2 shows aggregate numbers regarding the extracted feature vectors for the god classes.

The code for this step can be found in *extract_feature_vectors.py* file. The corresponding output file is named *filepath.csv* in the *results_csv* folder.

2 Clustering

2.1 Algorithm Configurations

Two algorithms were used for clustering: Agglomerative and K-Means. For Agglomerative clustering, the distance function used was Euclidean, and the linkage rule was Ward. In both the casses the feature vectors file created in the previous step was used as an input and the number of clusters was asked from the user as input. The code for this step can be found in *k_means.py* and *hierarchical.py* file. And the results of the clustering can be found in the *results_csv* folder and the nomenclature of the different results file is explained in the code. Both the files can be run by providing the path to the feature vectors file created in the Part-02 of this project present in the *results_csv* folder and the number of clusters as an argument to the script.

2.2 Testing Various K & Silhouette Scores

The next step is to test the clustering algorithms for various values of k. The results of the clustering are evaluated using the silhouette score. The silhouette score is a measure of how similar an object is to its cluster compared to other clusters. The silhouette score ranges from -1 to 1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. The code for this step can be found in *silhouette.py* file. To run the code, the user either can provide just the path to the feature vectors file as an argument to the script or can provide both the path to the feature vectors file and the path to the corresponding cluster file created in the last part as an argument to the script. In the first case, the script will run the clustering for the values of k from 2 to 59 and will calculate the corresponding silhouette score for each value of k. In the second case, the script will calculate the silhouette score for the given cluster file.

The output of the script is a file named *filename_silhouette_scores.csv* in the *results_csv* folder in the first case and in the second case it is printed on the

console.

The summarization of silhouette scores for the optimal number of clusters is as follows:

1. We found out that for the file *CoreDocumentImpl.csv* the optimal number of clusters in case of k-means increases with the increase in the number of clusters, so the maximum is achieved at k = 59, while in case of agglomerative clustering the optimal number of clusters is achieved at k = 45. The silhouette scores for the optimal number of clusters can be found in the *CoreDocumentImpl_silhouette_scores.csv* file in the *results_csv* folder.

2. For the file *DTDGrammar.csv* the optimal number of clusters in case of both the clustering algorithms is achieved at k = 58. The silhouette scores for the optimal number of clusters can be found in the *DTDGrammar_silhouette_scores.csv* file in the *results_csv* folder.

3. For the file *XSDIncludeHandler.csv* the optimal number of clusters in case of k-means is achieved in the beginning at k = 4, while in case of agglomerative clustering the optimal number of clusters is achieved at k = 2. The silhouette scores for the optimal number of clusters can be found in the *XSDIncludeHandler_silhouette_scores.csv* file in the *results_csv* folder.

4. For the file *XSDHandler.csv* the optimal number of clusters in case of both the clustering algorithms are achived at k = 2. The silhouette scores for the optimal number of clusters can be found in the *XSDHandler_silhouette_scores.csv* file in the *results_csv* folder.

The corresponding graphs for the silhouette scores for the optimal number of clusters can be found in the *results_csv* folder. The corresponding graphs for the silhouette scores for the clusters from k = 2 to 59, for the God Classes are shown in the fig.1

3 Evaluation

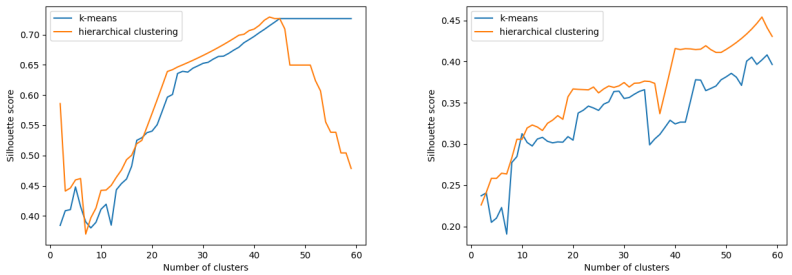
The evaluation of the models is done in two steps:

1. Defining the ground truth.

2. Computing the precision, recall scores and F1 scores for the optimal configurations found discussed below.

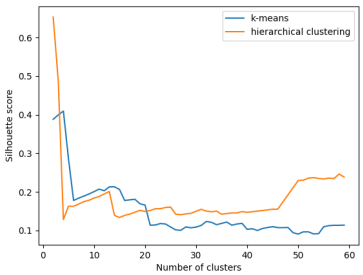
3.1 Ground Truth

As we are not the writers of the *xerces2-j-src* repository, we do not have the ground truth for the God classes in the repository. So, we created the ground truth by manually according to the keywords given in the Part-04 of the project slides. The keywords can be found in the *keywords.txt* file. To define the ground truth labels for each method of the God classes, we used the keywords present in

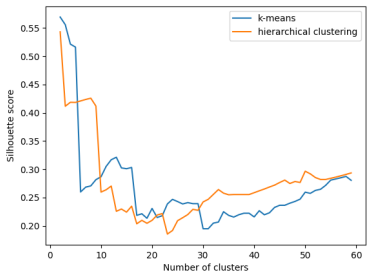


(a) CoreDocumentImpl

(b) DTDGrammar



(c) XIncludeHandler



(d) XSDHandler

Figure 1: Silhouette scores for the clusters from k = 2 to 59.

Class Name	Agglomerative		K-Means	
	Prec.	Recall	Prec.	Recall
CoreDocumentImpl	0.47	0.52	0.45	0.69
DTDGrammar	0.54	0.47	0.54	0.35
XIncludeHandler	0.66	0.46	0.67	0.77
XSDHandler	0.35	0.72	0.36	0.89

Table 3: Evaluation Summary

the *keywords.txt* file. If any of the keywords is present in the method name, then the method is labeled as according to that keyword and if multiple keywords are present in the method name then the first keyword determines the cluster label for that method. If none of the keyword is present in the method name, then the method is labeled as in '*other*' cluster. The cluster labeles are then converted into numerical values for the evaluation of the clustering algorithms.

Though the ground truth is not perfect. The drawback of the ground truth is that it is based on the keywords present in the method names and the keywords are not exhaustive, so the ground truth may not be perfect. And in the case of the multiple ground truth labels for a single method, we have taken the first keyword as the cluster label for that method, which may not be the correct way to label the method. Still, it is a good approximation of the actual ground truth

3.2 Precision and Recall

For calculating the precision and recall scores, we used the ground truth labels and the calculated cluster labels obtained from the clustering algorithms. The number of clusters in the clustering algorithms is set to the number of clusters in the ground truth labels. The code for this step can be found in the *prec_recall.py* file. We used the number of cluster labels in the ground truth instead of the optimal number of clusters give by the silhouette score because of in some of the God classes the silhouette score increases with the number of clusters and it wouldn't make sense to use for example, 5 clusters for the ground truth and 59 clusters for the clustering algorithms.

Precision and Recall, for the configurations found in Section 2, are reported in Table 3.

3.3 Practical Usefulness

The practical usefulness of the obtained code refactoring assistant in a realistic setting is that it can be used to identify the God classes in the repository and then can be used to refactor the code by breaking the God classes into smaller classes with single responsibility. This will make the code more maintainable and understandable.