IMA Project 1

Zhizi Wen

April 29, 2022

0 Code Repository

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

Repo: https://github.com/usi-ima/project-1-god-classes-ZhiziWen
Commit: f5b3eae

1 Data Pre-Processing

1.1 God Classes

The god classes I identified, and their corresponding number of methods can be found in Table 1.

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

Table 1: Identified God Classes

first, I looked for java file, by looking for file name containing "java". If the file name has "java", then I read the file and parse the file in a tree.

I created a list called "methods". In the tree, when I found a class using javalang.tree.ClassDeclaration, then I added the number of methods in this class in the list "methods". With the list of numbers of methods in each class, I calculated the average and the standard deviation of methods in a class with the list "methods", and got a threshold using the given formula. Then I compared each class with this threshold. If the number of methods in a class is higher than the threshold, than it is identified as a god class. I added the god class name as the key, and its number of method as the value into a dictionary called god_class, and then turned the dictionary into a dataframe.

1.2 Feature Vectors

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

Class Name	**# Feature Vectors**	**# Attributes***
CoreDocumentImpl	117	65
DTDGrammar	91	101
XSDHandler	106	181
XIncludeHandler	108	176

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

First, I defined four utility functions. With "get_fields" and "get_method", I went through all the fields and methods in a class and appended them in a list. With the function and "get_fields_accessed_by_method", I first created a list "members", and then checked whether a method contains fields, by checking if the type of node is MemberReference. If it is, then I appended the qualifier of the node to the list "members" when the qualifier is not empty; or appended the member of the node to the list "members" when the qualifier is empty. With the functions "get_methods_accessed_by_method" , I first created a list "methods", I went through all the nodes in the method tree and appended member of the node in the list "methods" if the node is a method invocation.

In main, I first parsed the file with a god class into a tree, and got fields and methods in the god class using the two utility functions "get_fields" and "get_method". I used them as the key of dictionaries "fields" and "methods".

Dictionary "fields": {field1: 0; field2: 0; field3: 0; ...}
Dictionary "methods": {method1: 0; method2: 0; method3: 0; ...}

I iterated over all methods in the god class. I used the function "get_methods _accessed_by_method" and "get_fields_accessed_by_method" to get all the methods and fields in the methods. Then I checked whether these fields and methods are in the class fields and methods, if they are, then I updated the dictionary fields and dictionary methods, i.e., if the key of the two dictionaries (method or field) is used in method, the value is updated to 1.

Dictionary "fields": { field1: 1; field2: 0; field3: 0 ... }
Dictionary "methods": { method1: 0; method2: 1; method3: 0 ...}

Then I appended these 2 dictionaries into another 2 dictionaries called "fields_list" and "methods_list", and reset the two dictionaries "fields" and "methods" so that they have only 0 as value and can join the next loop iteration. After iterating over all the methods in the god class, these 2 dictionaries "fields list" and "methods list" uses names of methods as key, and their feature vectors as value.

Dictionary "fields_list":
{method1: {field1 = 1; field2 = 0; field3 = 0; ... }
method2: {field1 = 0; field2 = 0; field3 = 0; ... }
method3: {field1 = 0; field2 = 0; field3 = 0; ... } }
Dictionary "methods_list":
{method1: {method1 = 0; method2 = 1; method3 = 0; ... }
method2: {method1 = 0; method2 = 0; method3 = 0; ... }
method3: {method1 = 0; method3 = 0; method3 = 0; ... } }

Then, I turned these 2 dictionaries into dataframe and concat them. After that, I kept only columns which have at least a non zero value. The feature vector csv file is created based on the final dataframe.

2 Clustering

2.1 Algorithm Configurations

In the two file "k_means.py" and "hierachical.py", I first defined a utility function called "CSV_reader" to read the feature vector csv file. I read the csv file into a dataframe, turned the first column of the dataframe, which is the name of the methods into a list, and turn the rest of dataframe which is feature vectors into a nested list. With this function, I returned a list of method name and a nested list of feature vectors.

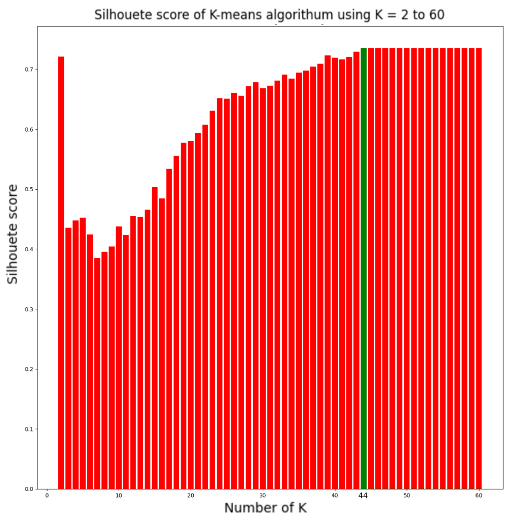
In K-means.py, I defined a function called "Kmeans_Calculator" which uses k-means algorithm. The k-means clustering algorithm uses the Euclidean distance to measure the similarities between objects. In this function, I used sklearn.cluster.kMeans. When I assigned the number of clusters K, sklearn.cluster. kMeans select the first K centroid in a smart way. To ensure reproducibility, i.e., always have the same first centriods, I used "random_state". Then the feature vectors will be assigned to a cluster which is closest to one of the K centroid.After that, the centroid of each clusters is recalculated, and the feature vectors will be reassigned to the cloest centroid. This will go on until centroids of every cluster do not change. Then, I got the cluster of each feature vector using kmeans.labels_, turn it into a dataframe, sort the dataframe with the clustering id and save it as a csv file.

In hierachical.py, I defined a function called "hierarchical_Caculator" which uses hierarchical, agglomerative algorithum. In this function, I used complete linkage rule. Complete-linkage measures distance between the farthest pair of data points in two clusters. I used complete linkage rule, because it usually produces tighter clusters than single-linkage. In this function, I used sklearn.cluster.AgglomerativeClustering. Agglomerative clustering starts with each data point as its own cluster. The two clusters that are closest to each other are combined into a one cluster. The next closest clusters are put together, and so on, until the complete data set is included in only one cluster. Then, I got the cluster of each feature vector using clustering.labels_,turn it into a dataframe, sort the dataframe with the clustering id and save it as a csv file.

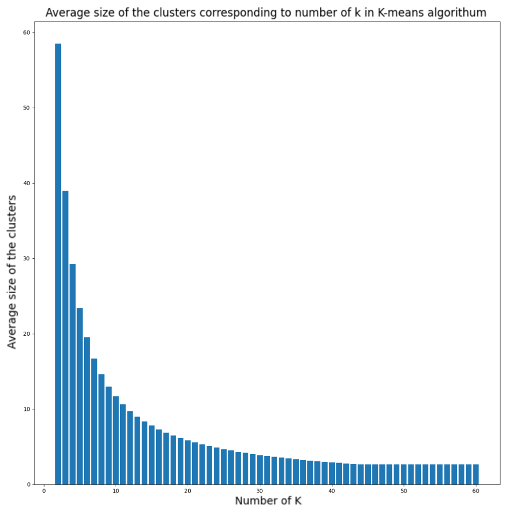
After defining these two function "CSV_reader" and "Kmeans_Calculator" or "hierarchical_Calculator", I used them with inputting the feature vector csv file and defining the number of clusters K.

2.2 Testing Various K & Silhouette Scores

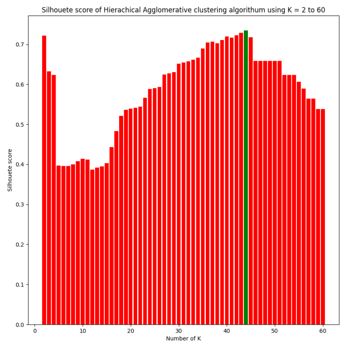
The following picture shows how the silhouete score changes with K = 2 to 60, and using K-Means algorithm to cluster the god class CoreDocumentImpl. The green bar stands for the highest silhouete score and the best K. The best K for Kmeans algorithum is 44 with Silhouete score of 0.7350427350427351. The code raises warning after K is larger than 44. I assume that this is because the data has not enough unique data points, and when K is larger than 44, the clustering will be the same as when K is 44.



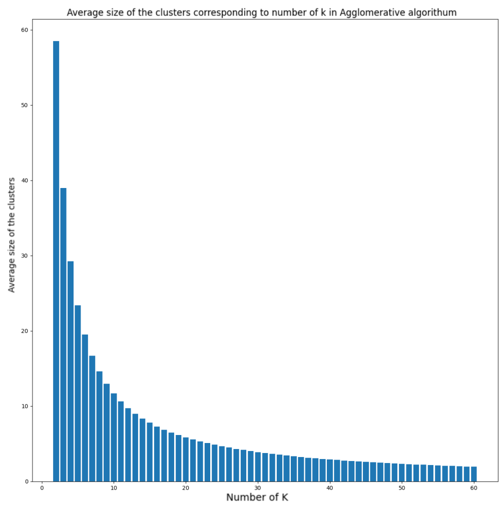
The following graph shows that the average size of clusters corresponding to the number of K in K-Means algorithm. We can see that the average size of clusters decreases with K increases from 2 to 44, and remain the same after K is larger than 44.



The following picture shows how the silhouete score changes with K = 2 to 60, and using Agglomerative algorithm to cluster the god class CoreDocumentImpl. The best K for Agglomerative algorithm is 44 with Silhouete score of 0.7350427350427351. The green bar stands for the highest silhouete score and the best K.



The following graph shows that the average size of clusters corresponding to the number of K in Agglomerative algorithm. We can see that the average size of clusters decreases with K increases.



In summery, the Agglomerative algorithm is better than K-Means algorithm in this case. When there is not enough unique data points, K-Means cannot keep on clustering the data when K is larger than the unique data points. So when we need to use a large K, Agglomerative algorithm is a better choice. The best K for every god class is listed in the following table 3.

Class Name	**Agglomerative**	**K-Means**
CoreDocumentImpl	44	44
DTDGrammar	55	59
XSDHandler	3	3
XIncludeHandler	2	2

Table 3: Best K with Agglomerative and K-Means algorithms

3 Evaluation

3.1 Ground Truth

I computed the ground truth using a self-defined command, by iterating over the method list and keyword list, and see if a method contain a keyword. I created a dictionary, which has the keywords as key. When a method contains a keyword, then the method will be appended as value of the corresponding key, and the iteration goes to the next method. If a method is not added because it does not contain a keyword, then it will be appended to the key ”None”. The generated files are checked into the repository with the names ”ground-truth.csv”.

The disadvantage of the given ground truth is that it has too little keywords. Using the god class CoreDocumentImpl, there are 117 methods in total and there are 69 methods which contain no keyword. This will create many nonsensical intra-pairs in the "None" clusters. However, for the methods which have a keyword, they do not contain 2 keywords at the same time. So the advantage is that the keyword classify every method into exactly one cluster.

3.2 Precision and Recall

Precision and Recall, for the optimal configurations found in Section 2, are reported in Table 4. Precision and Recall are rounded to 5 digits after the decimal point.

Class Name	**Agglomerative**		**K-Means**	
	Prec.	Recall	Prec.	Recall
CoreDocumentImpl	0.68098	0.31142	0.68098	0.31142
DTDGrammar	0.87975	0.06359	0.87770	0.05581
XSDHandler	0.37117	0.96132	0.37117	0.96132
XIncludeHandler	0.69583	0.95653	0.69583	0.95653

Table 4: Evaluation Summary

As we can see in the table, for the same class, the precision and recall are very similar when we cluster them in K-Means or Agglomerative algorithm. When precision is high, then recall is low and vice versa. Recall for DTDGrammar is very low, the reason might be that the ground truth given do not cover most of the methods. 64 of the 92 methods do not have a matching keyword, and they are classified as "None", causing a large number of intra-pair in the ground truth.

3.3 Practical Usefulness

In summary, this code can be useful for the first attempt of clustering, but programmers cannot blindly trust it. The code has got at least three advantages. First, the code classifies methods based on shared used fields or methods. This makes sense for us to put them into a class for reusability purposes. Second, the code can generate meaningful clusters much faster than manual labor. Third, the code can also help to find a god class quickly. There is also one disadvantage. In this code, the "best" K stands for the highest silhouette score. In reality, this might not be the best K when programmers split a god class. For example, with the K-Means clustering algorithm, class CoreDocumentImpl has the highest silhouette score when there are 44 clusters, but the second-highest silhouette score when there are only 2 clusters with a marginally lower silhouette score. For the actual usage, it might be better to split this god class into 2 classes. For example, it would be much easier to explain to a new programmer colleague the functionality of every class, when there are 2 classes instead of 44 classes.

However, the code doesn't realize the trade off between number of clusters and a higher silhouette score. For this reason, when programmers use the code, they should not just take the K with the highest silhouette score, but also take a look at the silhouette score of every k or take a look at the k with the top 3 silhouette score.