Information Modelling & Analysis – Project 1

Claudio Maggioni

Code Repository

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

::: center Repository: https://github.com/infoMA2023/project-01-god-classes-maggicl

Commit ID: **TBD** :::

Data Pre-Processing

God Classes

The first part of the project requires to label some classes of the *Xerces* project as "God classes" based on the number of methods each class has. From here onwards the Java package prefix `org.apache.xerces` is omitted when discussing fully qualified domain names of classes for sake of brevity.

Specifically, I label "God classes" the classes that have a number of methods six times the standard deviation above the the mean number of methods, i.e. where the condition

$$|M(C)| > \mu(M) + 6\sigma(M)$$

holds.

To scan and compute the number of methods of each class I use the Python library `javalang`, which implements the Java AST and parser. The Python script `./find_god_classes.py` uses this library to parse each file in the project and compute the number of methods of each class. Note that only non-constructor methods are counted (specifically the code counts the number of `method` nodes in each `ClassDeclaration` node).

Then, the script computes mean and standard deviation of the number of methods and filters the list of classes according to the condition described above. The file `god_classes/god_classes.csv` then is outputted listing all the god classes found.

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

Table 1: Identified God Classes

Class Name	**# Methods**
impl.xs.traversers.XSDHandler	118
impl.dtd.DTDGrammar	101
xinclude.XIncludeHandler	116
dom.CoreDocumentImpl	125

Feature Vectors

In this part of the project we produce the feature vectors used to later cluster the methods of each God class into separate clusters. We produce one feature method per non-constructor Java method in each god class.

The columns of each vector represent fields and methods referenced by each method, i.e. fields and methods actively used by the method in their method's body.

When analyzing references to fields, additional constraints need to be specified to handle edge cases. Namely, a field's property may be referenced (e.g. an access to array `a` may fetch its `length` property, i.e. `a.length`). In this cases I consider the qualifier (i.e. the field itself, `a`) itself and not its property. When the qualifier is a class (i.e. the code references a property of another class, e.g. `Integer.MAX_VALUE`) we consider the class name itself (i.e. `Integer`) and not the name of the property. Should the qualifier be a subproperty itself (e.g. in `a.b.c`, where `a.b` would be the qualifier according to `javalang`)

For methods, I only consider calls to methods of the class itself where the qualifier is unspecified or `this`. Calls to parent methods (i.e. calls like `super.something()`) are not considered.

The feature vector extraction phase is performed by the Python script `extract_feature_vectors.py`. The script takes `god_classes/god_classes.csv` as input and loads the AST of each class listed in it. Then, a list of all the fields and methods in the class is built, and each method is scanned to see which fields and methods it references in its body according to the previously described rules. Then, a CSV per class is built storing all feature vectors. Each file has a name matching to the FQDN (Fully-qualified domain name) of the class. Each CSV row refers to a method in the class, and each CSV column refers to a field, method or referenced class. A cell has the value of 1 when the method of that row references the field, method or class marked by that column, and it has the value 0 otherwise. Columns with only zeros are omitted.

Table 2 shows aggregate numbers regarding the extracted feature vectors for the god classes. Note that the number of attributes refers to the number of fields, methods or classes actually references (i.e. the number of columns after omission of 0s).

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

Class Name	**# Feature Vectors**	**# Attributes***
impl.xs.traversers.XSDHandler	106	183
impl.dtd.DTDGrammar	91	106
xinclude.XIncludeHandler	108	143
dom.CoreDocumentImpl	117	63

Clustering

In this section I covering the techniques to cluster the methods of each god class. The project aims to use KMeans clustering and agglomerative hierarchical clustering to group these methods toghether in cohesive units which could be potentially refactored out of the god class they belong to.

Algorithm Configurations

To perform KMeans clustering, I use the `cluster.KMeans` Scikit-Learn implementation of the algorithm. I use the default parameters: feature vectors are compared with euclidian distance, centroids are used instead of medioids, and the initial centroids are computed with the greedy algorithm `kmeans++`. The random seed is fixed to 0 to allow for reproducibility between executions of the clustering script.

To perform Hierarchical clustering, I use the `cluster.AgglomerativeClustering` Scikit-Learn implementation of the algorithm. Again feature vectors are compared with euclidian distance, but as a linkage metric I choose to use complete linkage. As agglomerative clustering is deternministic, no random seed is needed for this algorithm.

I run the two algorithms for all $k \in [2, 65]$, or if less than 65 feature vectors with distinct values are assigned to the god class, the upper bound of k is such value.

Testing Various K & Silhouette Scores

To find the optimal value of k for both algorithms, the distribution of cluster sizes and silhouette across values of k, and to apply the optimal clustering for each god class I run the command:

```
./silhouette.py --validate --autorun
```

Feature vectors are read from the `feature_vectors` directory and all the results are stored in the `clustering` directory.

Figures 1, 2, 3, and 4 show the distributions of cluster sizes for each god class obtained by running the KMeans and agglomerative clustering algorithm as described in the previous sections.

For all god classes, the mean of number of elements in each cluster exponentially decreases as k increases. Aside the first values of k for class `DTDGrammar` (where it was 2), the minimum cluster size was 1 for all analyzed clusterings. Conversely, the maximum cluster size varies a lot, almost always being monotonically non increasing as k increases, occasionally forming wide plateaus. The silhouette metric distribution instead generally follows a dogleg-like path, sharply decreasing for the first values of k and slowly increasing afterwards k. This leads the choice of the optimal k number of clusters for each algorithm to be between really low and really high values.

The figures also show the distribution of the silhouette metric per algorithm and per value of k. The optimal values of k and the respective silhouette values for each implementation are reported in Table 3.

From the values we can gather that agglomerative clustering performs overall better than KMeans for the god classes in the project. Almost god classes are optimally clustered with few clusters, with the exception of `CoreDocumentImpl` being optimally clustered with unit clusters. This could indicate higher cohesion between implementation details of the other classes, and lower cohesion in `CoreDocumentImpl` (given the name it would not be surprising if this class plays the role of an utility class of sort, combining lots of implementation details affecting different areas of the code).

Agglomerative clustering with complete linkage could perform better than KMeans due to a more urgent need for separation rather than cohesion in the classes that were analyzed. Given the high dimensionality of the feature vectures used, and the fact that eucledian distance is used to compare feature vectors, the hyper-space of method features for each god class is likely sparse, with occasional clusters of tightly-knit features. Given the prevailing sparsity, complete linkage could be suitable here since it avoids to agglomerate distant clusters above all.

Class Name	**KMeans K**	**KMeans silhouette**	**Hierarchical K**	**Hierarchical silhouette**
dom.CoreDocumentImpl	45	0.7290	45	0.7290
impl.xs.traversers.XSDHandler	2	0.5986	3	0.5989
impl.dtd.DTDGrammar	58	0.3980	2	0.4355
xinclude.XIncludeHandler	2	0.6980	2	0.6856

: Optimal hyperparameters and corresponding silhouette metrics for KMeans and Hierarchical clustering algorithm.

Evaluation

Ground Truth

I computed the ground truth using the Python script `./ground_truth.py` The generated files are checked into the repository with the names `clustering/{className}_groundtruth.csv` where `{className}` is the FQDN of each god class.

The ground truth in this project is not given but generated according to simple heuristics. Since no inherent structure or labelling from experts exists to group the methods in each god class, the project requires to label methods based on keyword matching whitin each method name. The list of keywords used can be found in `keyword_list.txt`. This approach allows to have a ground truth at all with little computational cost and labelling effort, but it assumes the method name and the chosen keywords are indeed of enough significance to form a meaningful clustering of methods that form refactorable cohesive units of functionality.

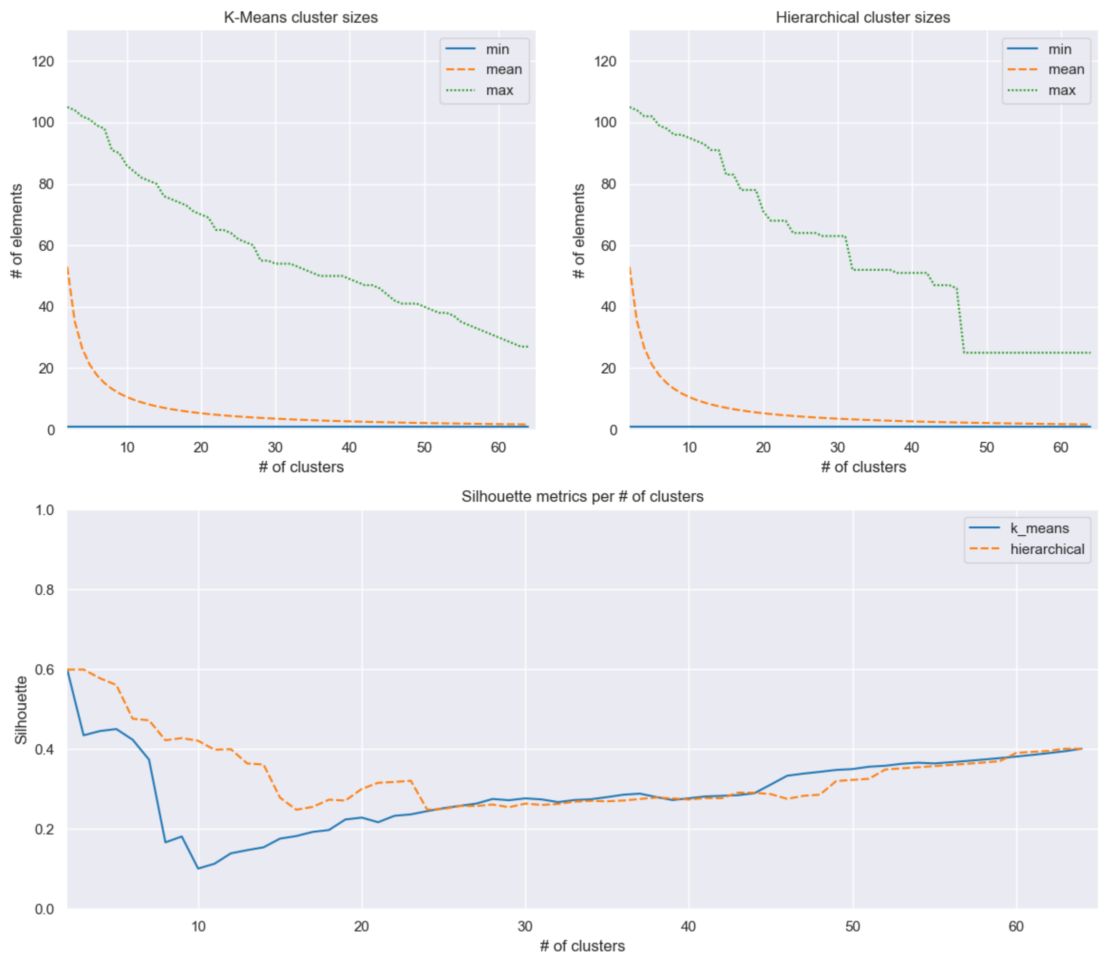

Figure 1: Clustering metrics for class impl.xs.traversers.XSDHandler

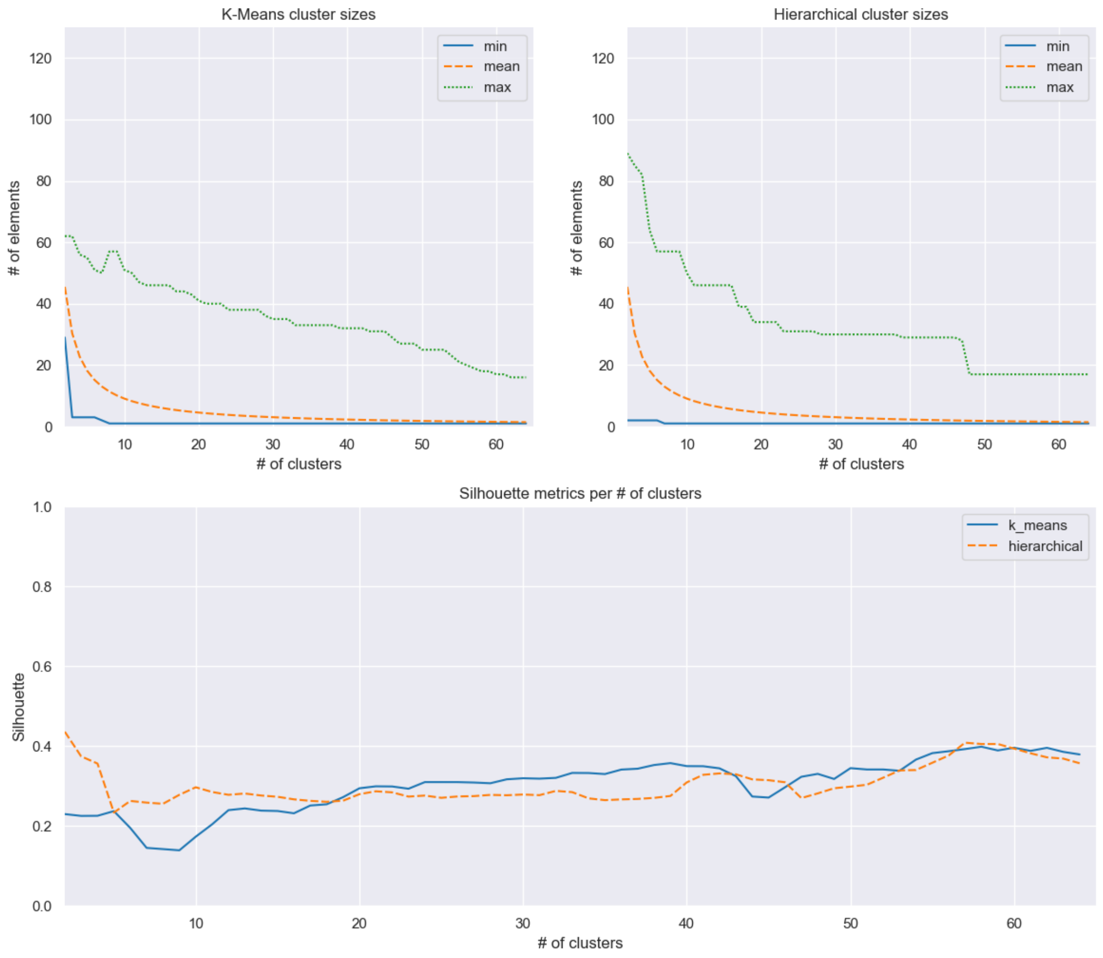

Figure 2: Clustering metrics for class impl.dtd.DTDGrammar

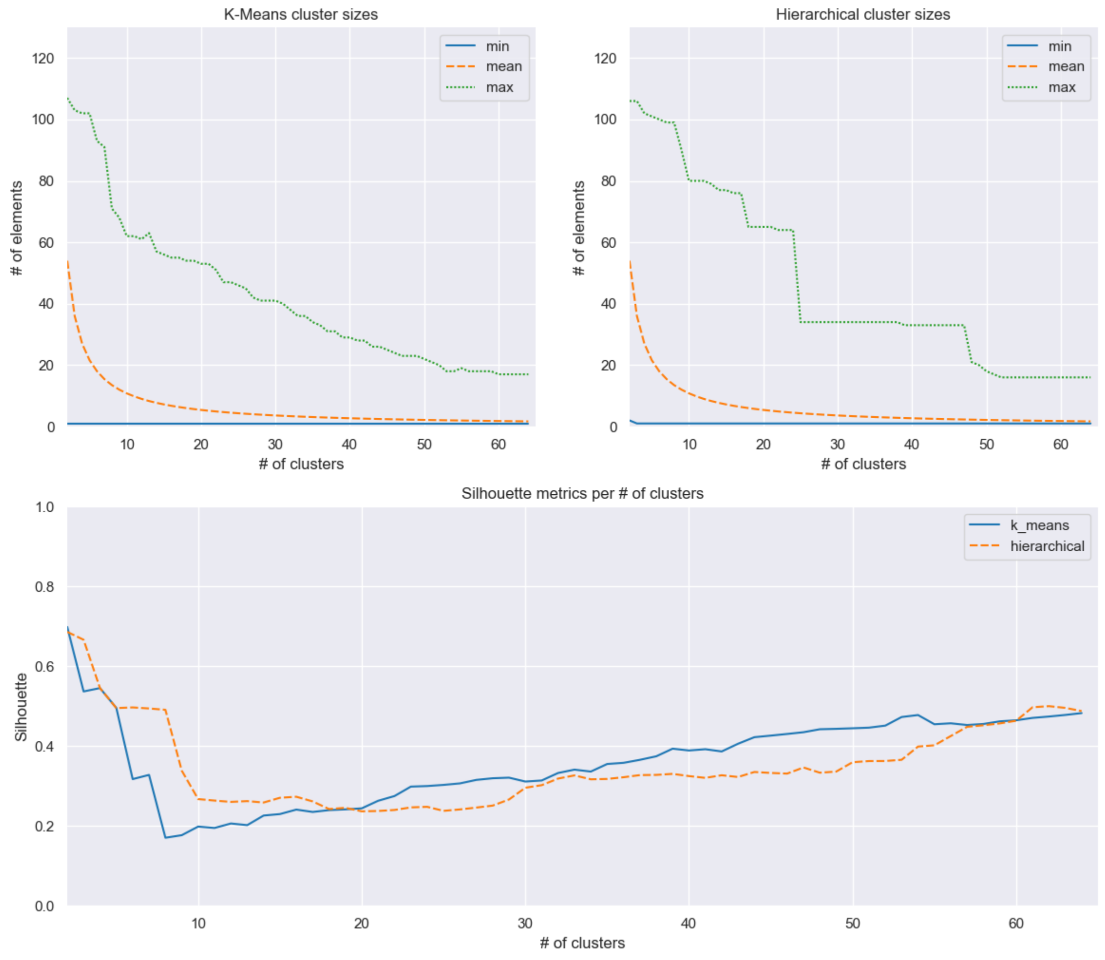

Figure 3: Clustering metrics for class xinclude.XIncludeHandler

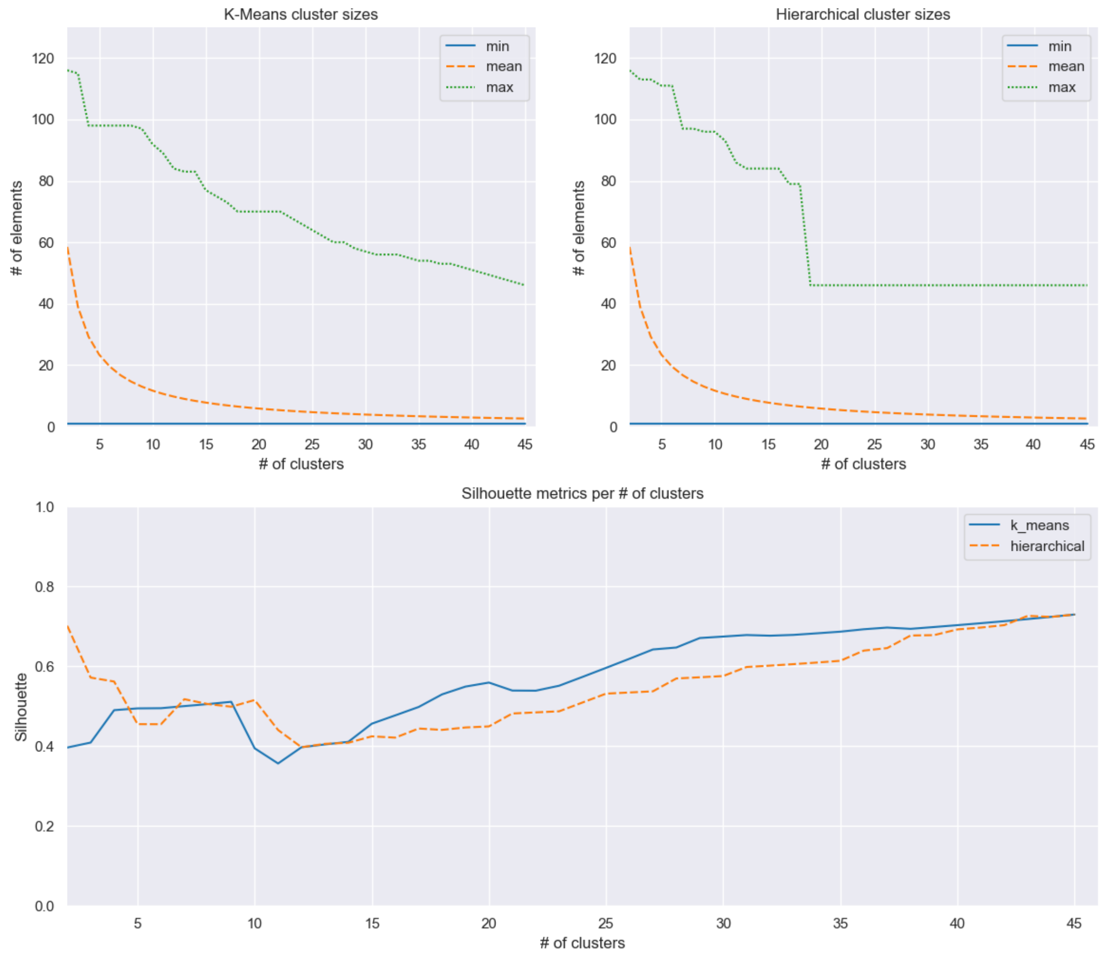

Figure 4: Clustering metrics for class dom.CoreDocumentImpl

Precision and Recall

Table 3: Evaluation Summary

Class Name	KMeans Precision	KMeans Recall	Agglomerative Precision	Agglomerative Recall
xinclude.XIncludeHandler	69.83%	97.80%	69.58%	95.65%
dom.CoreDocumentImpl	64.80%	28.26%	68.11%	29.70%
impl.xs.traversers.XSDHandler	36.17%	97.24%	36.45%	96.11%
impl.dtd.DTDGrammar	87.65%	6.87%	52.21%	94.28%

Precision and Recall, for the optimal configurations found in Section 3, are reported in Table 4.

comment precision and recall values

Practical Usefulness

Discuss the practical usefulness of the obtained code refactoring assistant in a realistic setting (1 paragraph).