IMA Project 1

Fabian Gobet

April 24, 2024

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-FabianGobet
Commit: b6aac9a

As a general overview, the code scripts can be located directly at the root of the project, whereas generated files from the tasks are in the `generated` folder. Furthermore, as an interactive way of seeing all the code working and the information generated for this report, a notebook file called `interactive_test.ipynb` may be found at the root of this repository.

1 Data Pre-Processing

1.1 God Classes

The identification of god classes is achieved by running the python script named `find_god_classes.py` . As a brief explanation, the code is fundamentally composed by two methods:

- `get_god_classes()`:
  This function takes a path to the src directory as input, generates a DataFrame containing class names and their respective method counts, identifies God classes based on a certain criteria, and returns a list of God classes.

- `generate_class_method_count_df()`:
  The search for all java files and method counts is done via the generate_class_method_count_df method, whereas the criteria applied to identify a god class is those that have a number of mthods above the sum of the mean and six times the standard deviation.

  The generate_class_method_count_df method does a walk through all the subpaths of the given input and for each .java file it parses it using the javalang library. After parsing it, it filters all nods from the AST by

classes, and for each it counts the number of methods in it applying the same transversal filtering technique.

This script is conveniently implemented to run on the command line, but it can also be imported and used in other scripts. To use in the command line one should structure the command as:

```
python find_god_classes.py [path to src directory]
```

By executing the former command, the results regarding the found god classes can be observed in Table 1.

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

Table 1: Identified God Classes

1.2 Feature Vectors

The implementation to extract all the features and feature vectors from each class can be found in `extract_feature_vectors.py`. As a brief explanation, the code is fundamentally composed by six methods:

- `get_fields()`:
  This method receives a *javalang.tree.ClassDeclaration* object as input and returns a set of all fields from this class.

- `get_methods()`:
  This method receives a *javalang.tree.ClassDeclaration* object as input and returns a set of all method names within the class.

- `get_fields_accessed_by_methods()`:
  This method receives a *javalang.tree.ClassDeclaration* object as input, filters that AST node by member reference and adds the name of the members to a set. If a qualifier is defined by the member, then the qualifier name is added instead. It then returns the set containing all the names.

- `get_methods_accessed_by_methods()`:
  This method receives a *javalang.tree.ClassDeclaration* object as input, filters that AST node by method invocations and adds the names to a set. It then returns the set contasining all the names

- `generate_feature_dataframe()`:
  This method receives as input a *javalang.tree.ClassDeclaration* object, a set of class method names and a set of class field names. It aggregates both sets into a list to form the feature space and creates a pandas data frame where the first column in the method name and the others the said features. For all methods, a row is created in the data frame. Then, an iterative process takes place for each method in the class declaration object, where the fields and methods accessed by that method are extracted and then assigned as a 1 in the respective row/column in the data frame. After this process, the data frame is returned.

- `extract_feature_vectors()`:
  This method receives a path to a java file, parses the java file using javalang and then generated a data frame for the feature vectors using all the prior functions. Any empty cells in the data frame are filled with 0. To finalize, the data frame is saved to csv format with the removal of the method name column.

This script is conveniently implemented to run on the command line, but it can also be imported and used in other scripts. To use in the command line one should structure the command as:

```
python extract_feature_vectors.py [path to java file] [optional:
    directory save path]
```

By importing `find_god_classes.py` we have access to all the found paths of the god classes. As such, by running the `extract_feature_vectors()` on each of these classes, four CSV files are generated containing the respective feature vectors, located in the `generated` folder. For in detail information about the features and feature vectors of each class, said files should be analyzed. A broad summary of each file can be found in Table 2

Class Name	**# Feature Vectors**	**# Attributes***
CoreDocumentImpl	117	139
DTDGrammar	91	166
XSDHandler	106	226
XIncludeHandler	108	200

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

2 Clustering

2.1 K-means

The implementation for k-means is straight forward and employs the use of the already implemented algorithm from `SKLearn` library. The `k_means_clustering()` function takes in as input a number of clusters and the path to the feature vector CSV. After having computed k-means, a CSV file is saved containing essentially two columns: the cluster ID and the method (vector) name.

2.2 Agglomerative

Similarly to k-means, the use of `SKLearn` library is employed, likewise receiving as input the number of clusters and the path to the feature vectors CSV, finally generating another CSV with cluster ID and respective method name (vector). Because linkage determines the clustering in agglomerative procedures, there is an extra input parameter which defines said property.

2.3 Algorithm Configurations

Regarding the algorithms configuration, the standard euclidean distance is used as distance function. Furthermore, because agglomerative procedures take in consideration the type of linkage in order to compute clustering, both complete and single linkage were tested.

2.4 Silhouette score

In order to compute the silhouette score, again, I made use of the `SKLearn` library for the `silhouette_score()` method. The implemented function `silhouette()` takes in as input the path of the feature vectors csv, an optional argument for the clustering CSV path, a max and a min number of clusters. The last two arguments are irrelevant if a clustering CSV path is given.

2.5 Testing Various K & Silhouette Scores

In order to test the various clustering configurations regarding the silhouette score, various iterations ranging from the number of clusters k = 2 until 60 were made for each of the classes. The information was then collected and used to generate the plots found in section A of the appendix.

We can observe that in all of the the classes the agglomerative clustering with single linkage rule always appears in the max value for the silhouette (noted as k, the dashed read line in the plots). This finding suggests then that for the classes XSDHandler, XIncludeHandler, DTDGrammar and CoreDocumentImpl the agglomerative clustering is most suited for a number of clusters of 2, 2, 2 and 45, respectively.

All the clustering csv files can be found with the generated/clusterings folder, whereas the silhouette scores and plots can be found within generated/silhouette_scores folder.

3 Evaluation

3.1 Ground Truth

The `get_ground_truth()` function in `ground_truth.py` takes as input a path to the feature vector csv file. It then iterates through all vectors from the csv, namely the method name, and adds each of these into a dictionary entry according to the containment of any of the keywords considering the order. In other words, after completion, the function has generated a dictionary, grouping method names by order of preference of the keywords list present in the repository. This dictionary is then used to generate a data frame and consequentially a csv file.

As a broad overview of the generated clustering we can look at the information in table Table 3.

Class Name	**# Clusters**	**# elements per cluster**
CoreDocumentImpl	11	[69,14,6,6,5,4,4,3,2,2,2]
DTDGrammar	5	[64,17,6,2,2]
XSDHandler	6	[90,9,4,2,2,1]
XIncludeHandler	5	[57,24,18,3,3,1]

Table 3: Table of ground truth quantities

This ground truth strategy is easy to implement and carries a logical semantic sense, that is we can semantically group samples in a feature space regarding keywords that describe the method or field. Naturally, we can also see how this strategy may be inappropriate because it doesn't necessarily translate into an optimal setting regarding our objective: to split god classes into sub classes,

grouping methods and fields according to their usage in the main class. It is difficult to come up with a rigorous ground truth that can effectively translate the problem into a solution, hence the current strategy employed for ground truth cluster generation should be seen as an heuristic.

3.2 Precision and Recall

In order to fully evaluate the former hypothesis regarding the ground truth, that is that the way we chose our ground truth may not correlate to solving the task at hand, I computed all precision and recall for each of the algorithms ranging from k=2 to 60. A special note regarding the optimal chosen configuration may be seen in the plots too. You can see the respective plots in section B of the appendix.

Considering the previously chosen optimal configuration we have the following values of precision and recall for each class in Table 4.

Class	**k**	**method**	**precision**	**recall**
CoreDocumentImpl	45	agglomerative single	0.68	0.31
DTDGrammar	2	agglomerative single	0.53	0.97
XSDHandler	2	agglomerative single	0.7	0.98
XIncludeHandler	2	agglomerative single	0.36	0.97

Table 4: Table of precision and recall for chosen optimal configuration

Nevertheless, by observation of the plots in section B of the appendix, we can immediately see the there might be different values of k of clustering methods that offer higher recall and precision. This observation is key and congruent with the hypothesis that our ground truth should be taken as an heuristic, rather than the actual ground truth.

3.3 Practical Usefulness

In a realistic setting, the previous procedure would systematically provide a way of refactoring classes that are too big and don't follow the OOP standards into smaller ones. As we seen before, the methodology is viable even if the viability of a ground truth is questionable. In other words, this procedure effectively provides a way of breaking classes that are too big into smaller ones whilst grouping functionalities (fields and methods) that are intrinsically correlated, thus allowing for a more OOP compliant structure.

Appendix

A 2.5 Testing Various K & Silhouette Scores

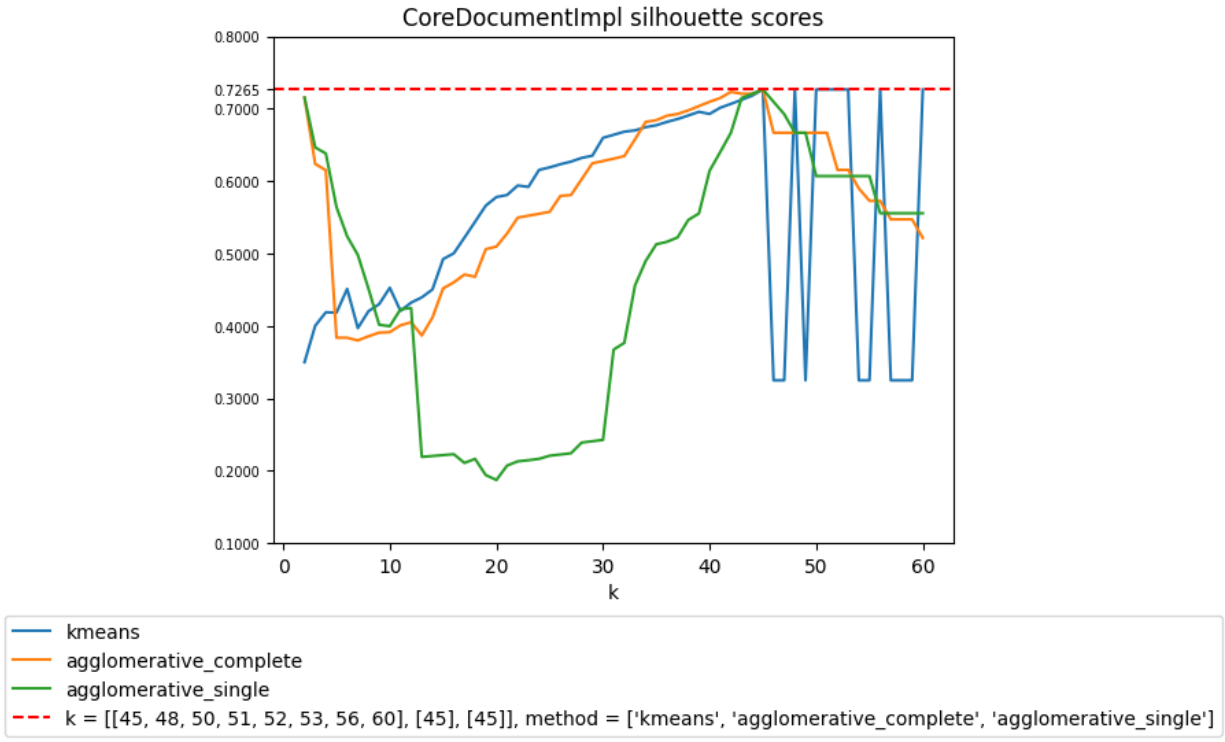

Figure 1: CoreDocumentImpl silhouete scores

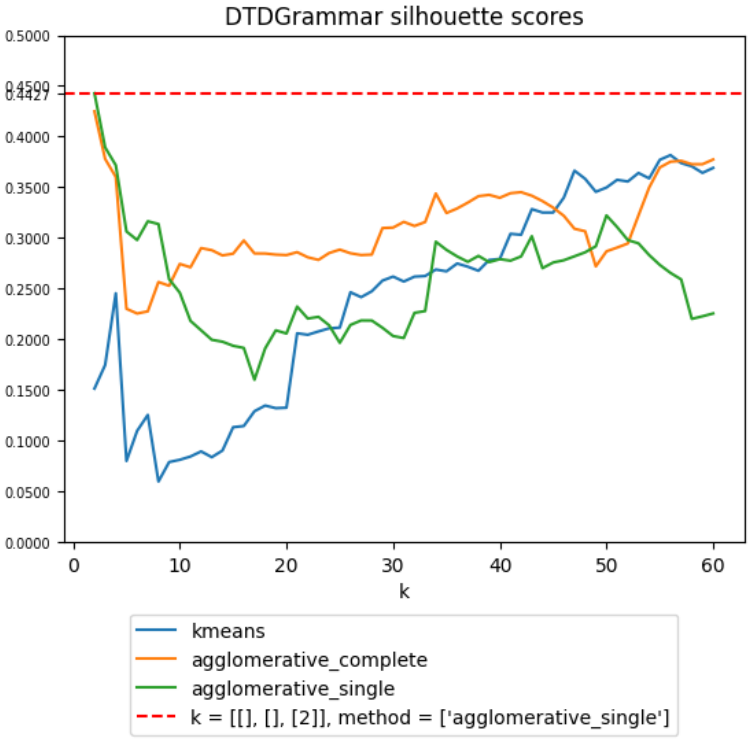

Figure 2: DTDGrammar silhouete scores

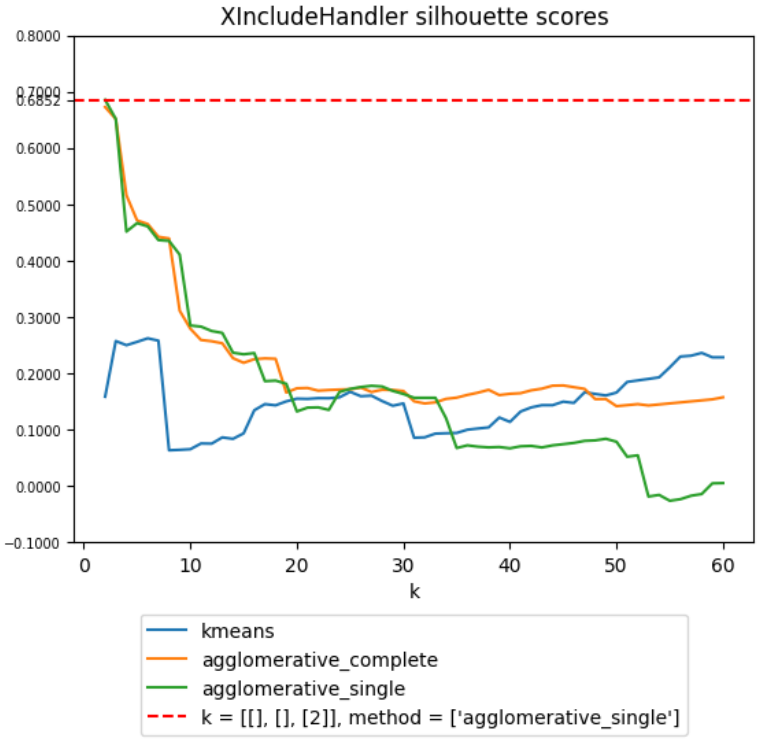

Figure 3: XIncludeHandler silhouete scores

Figure 4: XSDHandler silhouete scores

B Precision and Recall

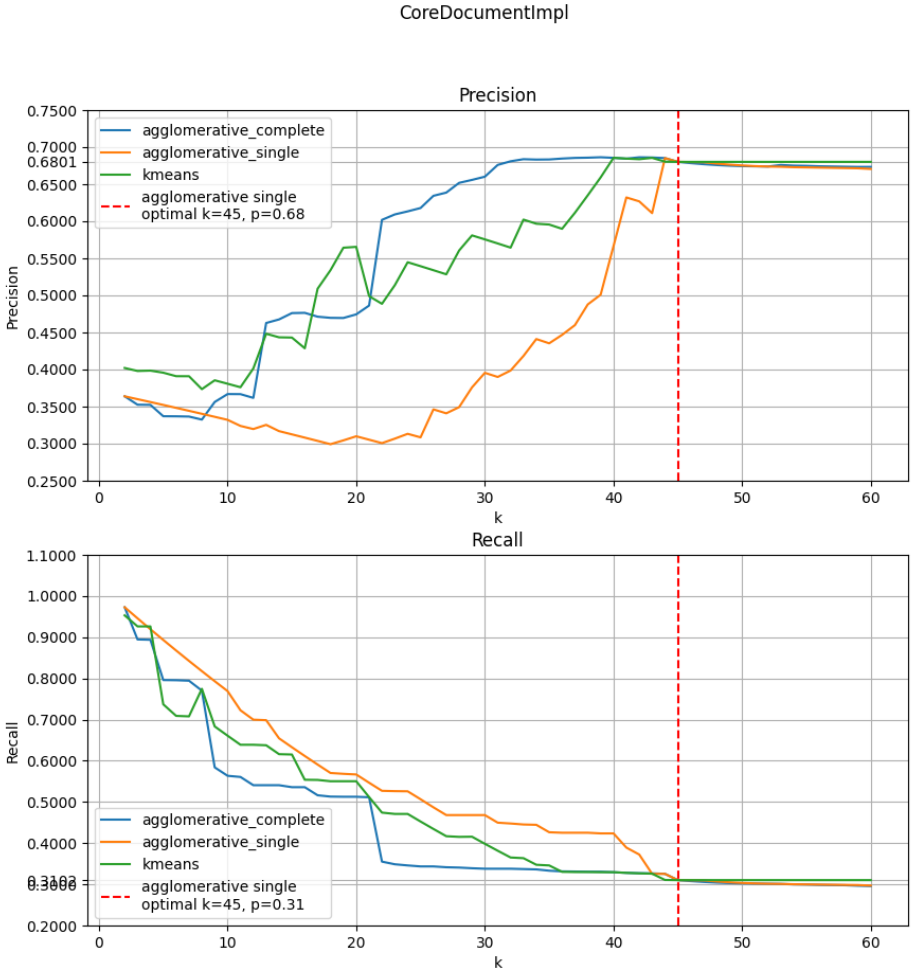

Figure 5: CoreDocumentImpl pr curves

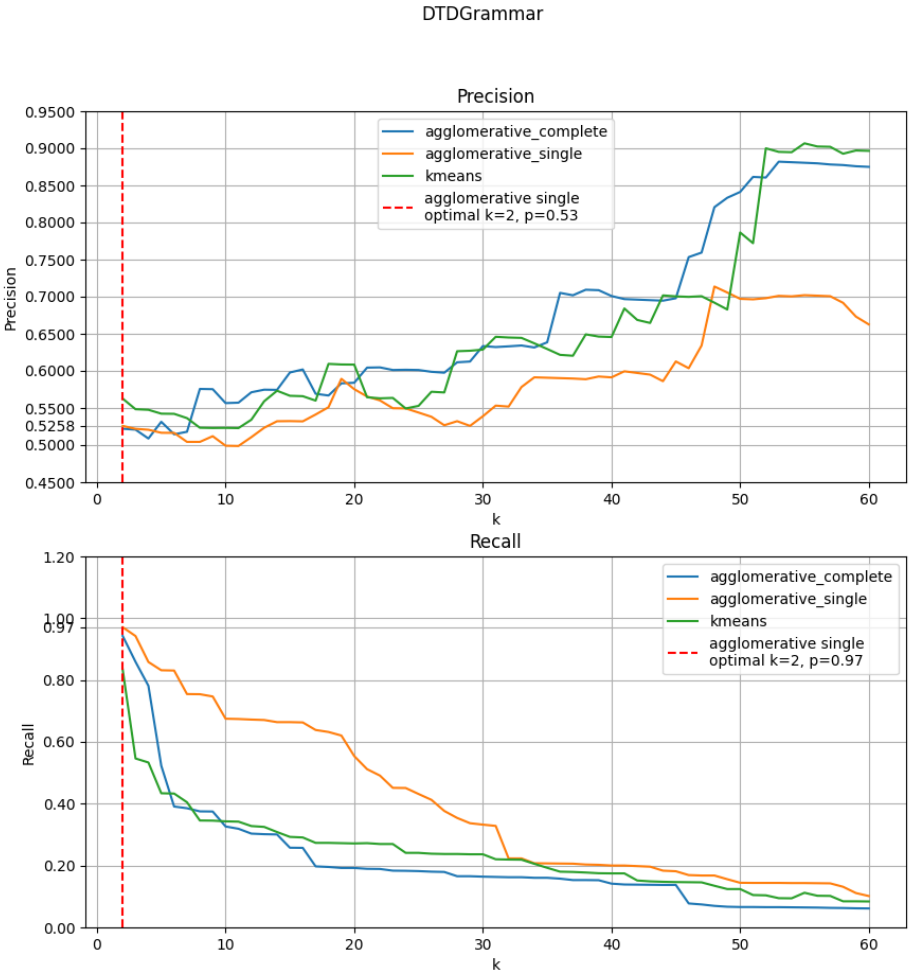

Figure 6: DTDGrammar pr curves

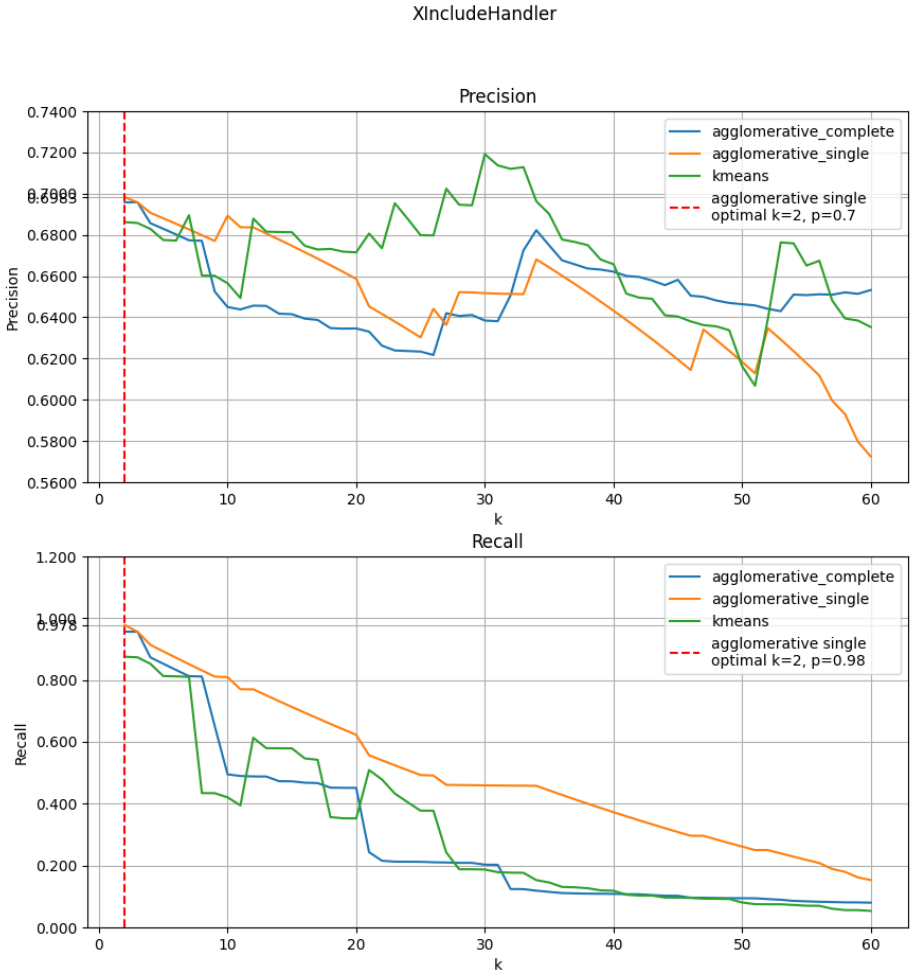

Figure 7: XIncludeHandler pr curves

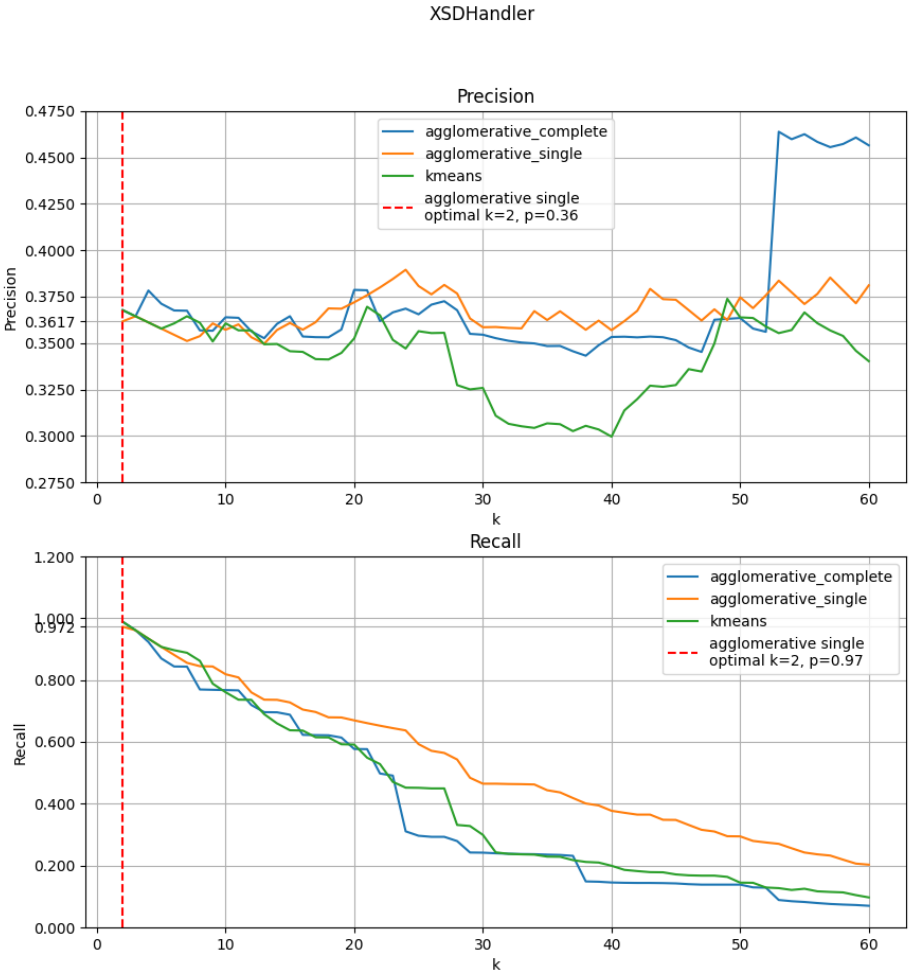

Figure 8: XSDHandler pr curves