

**Requirement 1**

The packages that use the DIP principle are the **edu.siu.sentise.factory** and **edu.siu.sentise** packages.

This is utilized the following way:

|  |  |
| --- | --- |
| **edu.siu.sentise.factory** | **edu.siu.sentise.preprocessing** |
| **High-level module:** POSTagProcessor | **High-level module:** SentiSE |
| **Interface:** WordChooser | **Interface:** TextPreprocessor |
| **Lower-level modules:** BasePOSUtility | **Lower-level modules:** AncronymHandler, QuestionMarkHandler,BIGramTriGramHandler,EmoticonProcessor,IdentifierProcessor,ExclamationHandler,URLRemover,ContractionLoader |

In **edu.siu.sentise.factory,** both **BasePOSUtility** and **POSTagProcessor** use WordChooser. POSTagProcessor uses BasePOSUtility to call the WordChooser interface:

if (!isPunctuation(word))

basePOSUtility.shouldInclude(leaf.label().toString(), word, pos, context, hashTable);

else

hashTable.put(leaf.label().toString(), word);

In **edu.siu.sentise.factory,** both **SentiSE** and **the various lower level modules** use TextPreprocessor. For example, in SentiSE calls the TextPrepocessor interface:

for (TextPreprocessor process : preprocessPipeline) {

sentimentDataList = process.apply(sentimentDataList);

}

**Requirement 2**

* **Which classes have poor cohesion?**

High Lack of Cohesion: SentiSE

Medium-High: MLPClassifier, NeuralConnection, POSTagProcessor

* **Which classes are highly coupled with other classes?**

Very High: SentiSE

Medium-High: MLPClassifier, WekaClassifierBuilder

* **Explain how would you redesign the program to remove the problems with coupling and cohesions?**

The SentiSE class is very large and contains many methods. There are multiple blocks of code similar to the one pictured below that calls the same function. It would be best to break up functions listed in this big block of code. The lack of cohesion makes it harder to work with the elements within this method. This code block (isCommandLineParsed) is very large and should probably be broken up into two or three methods.

The code above can be improved by substituting the algorithm.

private String totalAverage(ArrayList<CrossValidationResult> cvResults) {

double[] results = new double[11];

for (CrossValidationResult result : cvResults) {

String[] splits = result.toString().split(",");

for (int i = 0; i < splits.length; i++)

results[i] += Double.*parseDouble*(splits[i]);

}

for (int i = 0; i < results.length; i++)

{

results[i] /= 10;

String res = "";

{

if (i > 0)

res += ",";

res += results[i];

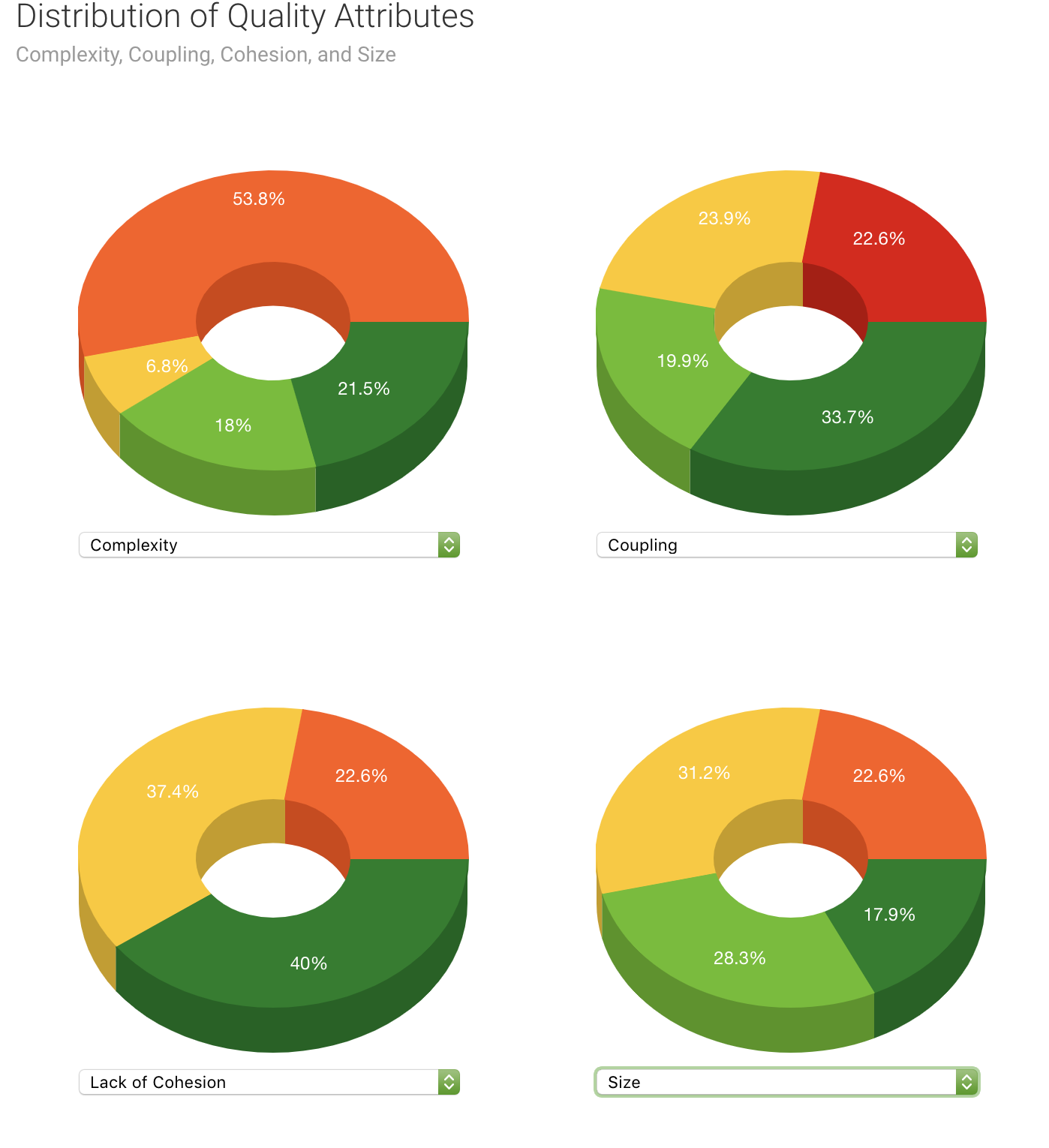
}

return res;

}



The code block in MLPClassifier above calls different methods and functions while setting inst equal to the output of those functions. Since this method is so closely tied to many different methods, it makes it harder to work with separately. It may be best to extract variables or methods from this function so that it won’t be so tightly coupled. **Among the more than 40 different metrics reported by CODEMR, pick four metrics that you believe are the most useful to identify problems in the design of a software. Briefly, explain your choices.**



Complexity: Complexity is important to software design because you do not want a method to be overly complex. If a method is too complex, future developments in the code may be difficult to perform. If someone else takes over a code that is not as experienced with it or you are revisiting a code, it may be difficult to determine how functions are performed if the code is too complex.

Coupling: When one or more methods are tightly coupled, they are harder to work with separately. If a section of code needs to be refactored, it would be very difficult to do so if it was bond to another code tightly. This metric allows you to see how efficient you code is in regards to coupling.

Lack of Cohesion: Cohesion represents the clarity of how modules are related. Cohesion works best when functions are combined in a method that work together to perform a single function. Seeing how closely your methods work together allows for a comprehendible code that will benefit any future refactor in the long run.

Size: When methods are too big, they become harder to work with. This ties into complexity and coupling as well. A methods that is too large can be overwhelming for a software engineer to work with. Methods may have to be extract or complicated expression may have to be condensed in order to make the code easier to work with.