Machine Configuration for the experiments ----

OS – Ubuntu 20.04 64 bit kernel version 5.11.0-25-generic

CPU – AMD Ryzen 5 4600H 3.0 Ghz 6 Cores 12 Threads

RAM – 8 GB

Response time for both Python and Java has been measured using pythons time.time() function (diffrence b/w start time and end time taken per file and then average is taken).

JAVA

|  |  |  |  |
| --- | --- | --- | --- |
| Tool | Input Form | Average Response Time (in seconds) | Storage requirement (in MBs) |
| BloatLibD | .jar file | 12.0431 | 1.52 |
| DepClean | Maven project | 4.4578 | Not Applicable |
| Jingredients | .jar file | 0.0001 | 2.2\* |

For DepClean, the Response Time used to calculate the Average Response Time involves only the Running Time of the actual Depclean dependency analysis on the maven project.

As DepClean relies on analysis of code rather than comparison with a database of libraries , storage requirement is not applicable to it.

Jingredients uses a database to compare class signatures to detect reuse. The database that we used was built from 214 .jar files out of which 192 .jar files were actually used by the tool to form the database. The storage utilized in the original paper from which Jingredients was implemented was 1.0 GB.

5 maven projects have been selected, and the used dependencies in them (whether direct or transitive) have been used as data (in the form of jars) for BloatLibD.

There are 38 jar files (taking union of all the dependencies as there were some common dependencies amongst them) in all. The 5 maven projects are converted to maven jars using the maven command line tool and these 5 projects are then presented as target input to the Jingredients tool.

In summary, BloatLibD runs on the 38 jar files (which were detected as dependencies by DepClean). The 5 maven projects are given as input to the DepClean tool and the .jar files from these 5 maven projects act as input to Jingredients.

Limitations of Jingredients – This tool needs high performance hardware and sufficiently large database to obtain good results. In the original paper, a high end workstation with 2 Six core processors and 64 GB of RAM was used alongside with a large corpus size of 1 GB ( which itself is constructed from an original repository data of size 77.8 GB with a total of 172,232 .jar files).

In our experiments with a much smaller database and modest machine configuration, the tool was unable to detect any instances of reuse , though it is able to identify the classes within the jar files of the maven projects.

PYTHON

|  |  |  |  |
| --- | --- | --- | --- |
| Tool | Accuracy (%) | Average Response Time (in Milli Seconds) | Storage requirement (in MBs) |
| PyCln | 99.29 | 0.1755 | Not Applicable |
| AutoFlake | 99.29 | 0.2280 | Not Applicable |

Both PyCln and AutoFlake rely on code analysis rather than matching with a white list, hence Storage requirement is not applicable on them

PyCln is executed as:

pycln -a -d ‘file location without quotes’

Autoflake is executed as :

autoflake --remove-all-unused-imports --recursive ‘file location without quotes’

3 Projects have been selected from GitHub at random consisting of 48 .py files (including init.py files as well)

To calculate metrics, total number of imports are obtained from all .py files and then based on whether an import is actually used and whether the tool correctly removes/flags the unused ones the following are defined:

AU – Import actually used

AUN – Import actually unused

DU – Import detected as used (better say not flagged as unused) by the tool

DUN – Import detected as unused by the tool

Total imports = 283 (each individual module imported is consider.

Consider the statement “ import sys,math,time “. This statement is equivalent to 3 separate imports

AU/DU ---> a True Positive (detected and is actually used in the file)

AU/DUN ---> a False Negative (actually used but incorrectly flagged as unused)

AUN/DU ---> a False Positive (actually unused but incorrectly missed by the tool)

AUN/DUN ---> a True Negative (actually unused and correctly flagged as unused)

In essence a confusion matrix is formed

For PyCln

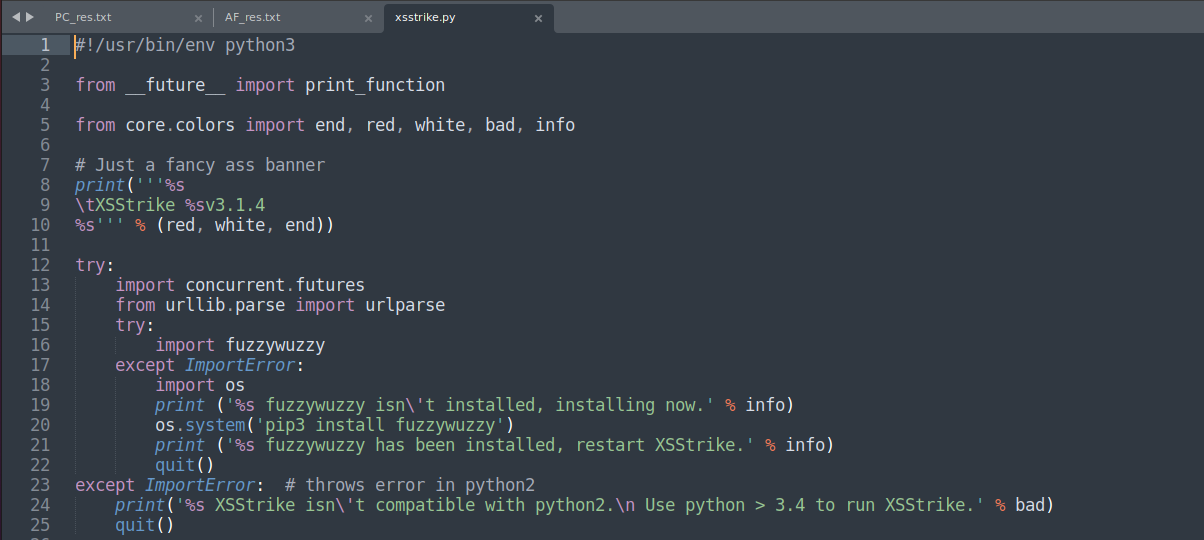
|  |  |  |
| --- | --- | --- |
| Tool | DU | DUN |
| AU | 259 | 0 |
| AUN | 2 | 22 |

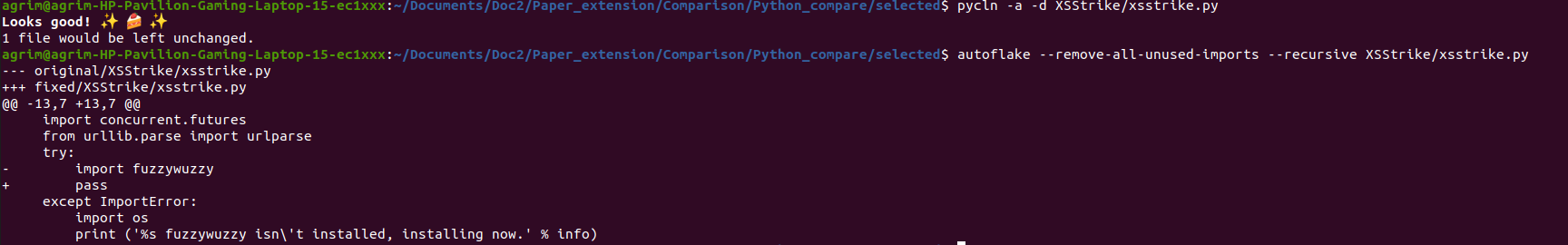
For AutoFlake

|  |  |  |
| --- | --- | --- |
| Tool | DU | DUN |
| AU | 258 | 0 |
| AUN | 2 | 23 |

Accuracy = TP+TN/ Total Obserations

\*\*\*\* Both produce near identical results but there are differences. One example is of the case of the file XSStrike/xsstrike.py. Particularly, the line 16 of this file is treated differently by AutoFlake and PyCln.

 Image from XSStrike/xsstrike.py. Line no. 16 is the focus.



Output of PyCln and AutoFlake differ in this case, as is evident from above image. PyCln does not flag the import at line 16 as unused while AutoFlake flags it.

\*\*\*\* Both the tools miss the case where when a file A.py imports from B.py as:

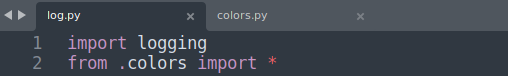
from B import \*

and not all the modules imported in B are used up by A.

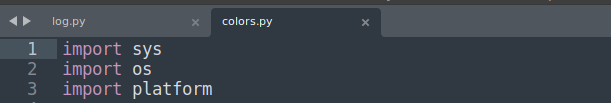
Case in point XSStrike/core/log.py file’s imports. The log.py file imports all modules from colors.py in the same directory (by from.colors using \*).

Now colors.py itself imports and uses the modules **sys**,**os** and **platform**

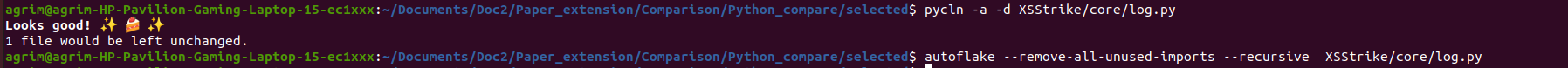
However, logs.py uses only the sys module imported from colors.py . Therefore despite 3 imports only 1 is used , 2 are unused, but both the tools don’t flag this.



Log.py’s imports, only sys module is directly utilized (exactly at line 162 and 167 of the file).



colors.py’s imports , all of these are used.



Both PyCln and AutoFlake don’t flag the imports in log.py