



SOEN 6611 (SOFTWARE MEASUREMENT)

CONCORDIA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING

METRICSTICS

Team H:

Naman Kumar
Yvonne Chooi Mei Lee
Yang Liu
Jothi Basu Lkv
Nasrin Maarefi

Professor:
Prof. PANKAJ KAMTHAN

Contents

1	Deliverable 2 (D2)	2
1.1	Problem 3: Effort Estimation	2
1.1.1	Effort estimation using Use Case Points (UCP) approach	2
1.1.2	Effort estimation using Basic COCOMO Model	4
1.1.3	Effort Estimate Difference between using UCP Approach, COCOMO and Actual Effort	4
1.2	Problem 4: Implementation of METRICSTICS	5
1.2.1	Tech Stack	5
1.2.2	Tests	7
1.3	Problem 5: Cyclomatic Number	12
1.3.1	Calculating the Cyclomatic Number (Cyclomatic Complexity)	12
1.3.2	Qualitative Conclusions	14
1.4	Problem 6: Object-oriented Metrics	16
1.4.1	Calculating the Object-Oriented Metrics - WMC, CF & LCOM	16
1.4.2	Qualitative Conclusions	18
1.5	Problem 7: Physical and Logical SLOC	20
1.5.1	Calculating the Physical SLOC and Logical SLOC	20
1.5.2	Qualitative Conclusions	20
1.6	Problem 8: Establishing Relation between Logical SLOC and WMC	21
1.6.1	Scatter Plot	21
1.6.2	Correlation Coefficient	22

Deliverable 2 (D2)

1.1 Problem 3: Effort Estimation

1.1.1 Effort estimation using Use Case Points (UCP) approach

We only have one actor (complex one) who interacts with the system using graphical user interface to do use cases, including import data (simple), generate data (simple) and calculate statistics measurements (average).

1. Calculating Unadjusted Actor Weight(UAW)

$$UAW = 3 \quad (1.1)$$

2. Calculating Unadjusted Use Case Weight (UUCW)

$$UUCW = 2 * 5 + 10 \quad (1.2)$$

$$= 20 \quad (1.3)$$

3. Calculating Technical Complexity Factors (TCF)

TCF Type	Factor Name	Weight	Perceived Impact Factor
T1	Distributed System	2	0
T2	Performance	1	3
T3	End User Efficiency	1	3
T4	Complex Internal Processing	1	3
T5	Reusability	1	3
T6	Easy to Install	0.5	5
T7	Easy to Use	0.5	5
T8	Portability	2	5
T9	Easy to Change	1	3
T10	Concurrency	1	0
T11	Special Security Features	1	0
T12	Provides Direct Access for Third Parties	1	0
T13	Special User Training Facilities are Required	1	0

Table 1.1: Technical Complexity Factors (TCF) and their Perceived Impact Factors

Perceived Impact Factor:

- (a) 0: No influence
- (b) 3: Average influence
- (c) 5: Strong influence

$$TCF = 0.6 + (0.01 * (2 * 0 + 1 * 3 + 1 * 3 + 1 * 3 + 1 * 3 + 0.5 * 5 + 0.5 * 5 +$$
 (1.4)

$$2 * 5 + 1 * 3 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0))$$

$$= 0.6 + 0.01 * 30$$
 (1.5)

$$= 0.9$$
 (1.6)

4. Calculating Environmental Complexity Factor (ECF)

ECF Type	Description	Weight	Perceived Impact Factor
E1	Familiarity with the use case domain	1.5	3
E2	Part-time workers	-1	3
E3	Analyst Capability	0.5	5
E4	Application Experience	0.5	5
E5	Object-oriented experience	1	5
E6	Motivation	1	5
E7	Difficult Programming Language	-1	1
E8	Stable Requirements	2	5

Table 1.2: Environmental Complexity Factors (ECF) and their Perceived Impact Factors

Perceived Impact Factor (when positive weights):

- (a) 0: No influence
- (b) 1: Strong, negative influence
- (c) 3: Average influence
- (d) 5: Strong, positive influence

Perceived Impact Factor (when negative weights):

- (a) 0: No influence
- (b) 1: Strong, favourable influence
- (c) 3: Average influence
- (d) 5: Strong, unfavourable influence

$$ECF = 1.4 + (-0.03 * (1.5 * 3 + (-1 * 3) + 0.5 * 5 + 0.5 * 5 + 1 * 5 +$$
 (1.7)

$$1 * 5 + (-1 * 1) + 2 * 5))$$

$$= 1.4 + (-0.03 * 25.5)$$
 (1.8)

$$= 0.635$$
 (1.9)

5. Calculating Unadjusted Use Case Points (UUCP)

$$UCP = UUCP * TCF * ECF$$
 (1.10)

$$= (UAW + UUCW) * TCF * ECF$$
 (1.11)

$$= (3 + 20) * 0.9 * 0.635$$
 (1.12)

$$= 13.1445$$
 (1.13)

6. Effort Estimation

$$EffortEstimate = UCP * ProductivityFactor$$
 (1.14)

$$= 13.1445 * 20$$
 (1.15)

$$= 262.89person - hour$$
 (1.16)

1.1.2 Effort estimation using Basic COCOMO Model

*a1 is usually 2.4 and a2 is 1.05 for projects with less than 50 KLOC Total lines of code = 0.166 KLOC

$$EffortEstimate = a1 * (KLOC)^{a2} \quad (1.17)$$

$$= 2.4 * 0.166^{1.05} \quad (1.18)$$

$$= 0.364 person - months \quad (1.19)$$

1.1.3 Effort Estimate Difference between using UCP Approach, COCOMO and Actual Effort

To compare, first we convert the COCOMO model estimation to person-hours. To do this, we suppose each person works 160 hours each month.

$$0.364 \text{ person-months} = 58.24 \text{ person-hours}$$

Comparing this with the effort estimate from the UCP approach (Use Case Points), which is 262.89 person-hours, we can observe a significant difference between the two estimates.

The Basic COCOMO model (Constructive Cost Model) provides a much lower estimate, potentially due to the simplicity of the model. In the UCP approach, various factors, including technical and environmental factors are assumed for estimation. Additionally, this approach differentiates between actors and use cases. However, in the COCOMO model, only the lines of code are considered irrespective of the logic and complexity of those codes for each use case.

It's essential to ensure that the same parameters and conditions are considered when comparing different models for project effort estimation.

Real effort: 42 hours

The difference in effort estimates from UCP and COCOMO compared to the actual effort arises from several factors. UCP's subjectivity in assessing complexities and changing project requirements can lead to inaccuracies. Human judgment in UCP, prone to biases and expertise levels, also affects estimates. External unpredictable abilities and the team's skill and efficiency further contribute to the variance. Additionally, UCP's assumption of a traditional development approach might not fit all projects, especially those using Agile methodologies. Finally, the effectiveness of project management significantly influences the alignment of actual effort with estimates. These elements collectively explain the discrepancies between the estimated and actual efforts.

1.2 Problem 4: Implementation of METRICSTICS

1.2.1 Tech Stack

To implement METRICSTICS, Python was used as the programming language, specifically Python 3. It is dynamically typed and garbage-collected. It is also able to support multiple programming paradigms, such as structured (procedural), object-oriented and functional programming. [1] For the development of this application, the object-oriented programming paradigm was used.

For the GUI, a library called Tkinter was used, which is a Python binding to the Tk GUI toolkit, Python's de facto standard GUI. [3]

To calculate descriptive statistics, users of METRICSTICS are able to either:

1. Generate random values
2. Upload a CSV file

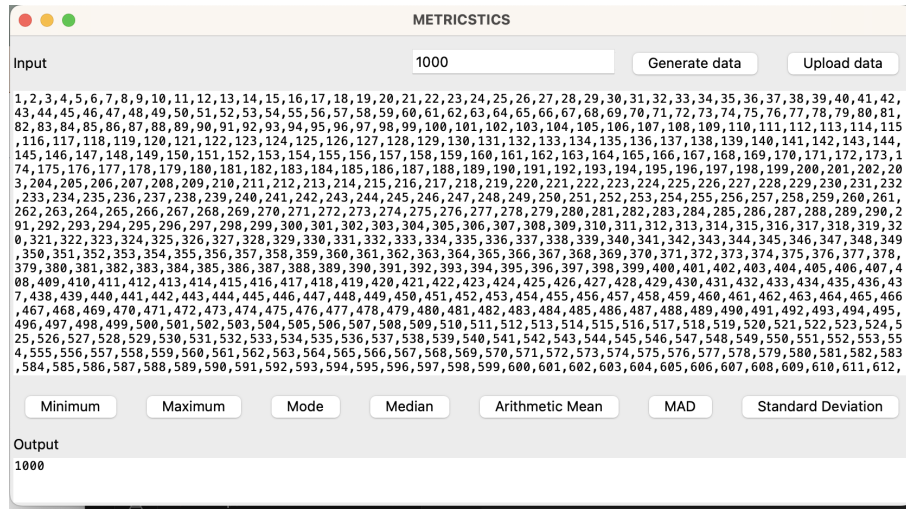


Figure 1.1: METRICSTICS Application

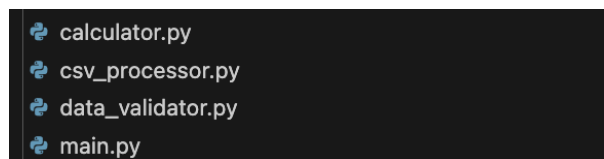


Figure 1.2: Python Classes

```

1  class Calculator:
2      def __init__(self, data):
3          self.data = data
4
5      def set_data(self, data):
6          self.data = data
7
8      def clear_data(self):
9          self.data = None
10
11     def is_data_exists(self):
12         if self.data == None or len(self.data) == 0:
13             raise Exception("No values to calculate")
14
15     # get the min value from dataset
16     def get_min(self):
17         self.is_data_exists()
18         min = self.data[0]
19         for value in self.data:
20             if value < min:
21                 min = value
22         return min
23
24     # get the max value from dataset
25     def get_max(self):
26         self.is_data_exists()
27         max = self.data[0]
28         for value in self.data:
29             if value > max:
30                 max = value
31         return max
32
33     # get the mode (value that appears most frequently) in dataset
34     def get_mode(self):
35         self.is_data_exists()
36         dict = {}
37         count, mode = 0, ''
38         for item in reversed(self.data):
39             dict[item] = dict.get(item, 0) + 1
40             if dict[item] >= count :
41                 count, mode = dict[item], item
42         return str(mode) + ": " + str(count) + " count(s)"

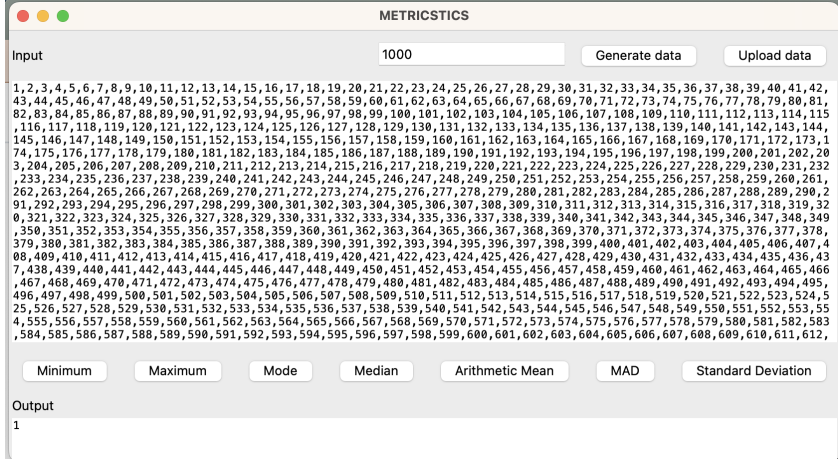
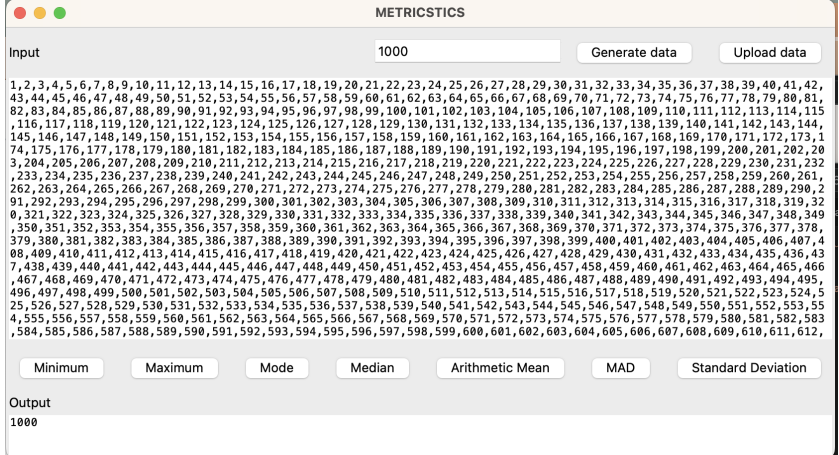
```

Figure 1.3: Sample Code - Object-oriented Paradigm

1.2.2 Tests

Input files that have been used for the tests can be found in the 'input' folder in the repository, while the output files of the corresponding inputs can be found in the 'output' folder. The examples below provide a summary of tests performed. Detailed and more extensive tests can be found in the repository folder.

- Input files with the prefix 'input{x}_{xxx}.csv' were used to test the csv upload function (labelled as the 'Upload data' button)
- Input files with the prefix 'random{x}.csv' were values generated from the button labelled as 'Generate data'. These values were copied into a csv file for tracing and verification purposes for the input and their corresponding outputs produced.
- Output files are prefixed with the input file name and the functions they have performed so that they can be easily identified and verified.
- For example, the 'Mininum' screenshot for 'random1.csv' is named as 'random1_min.png', the 'Standard Deviation' for 'input1_sequential.csv' is named as 'input1_standarddeviation.png', etc

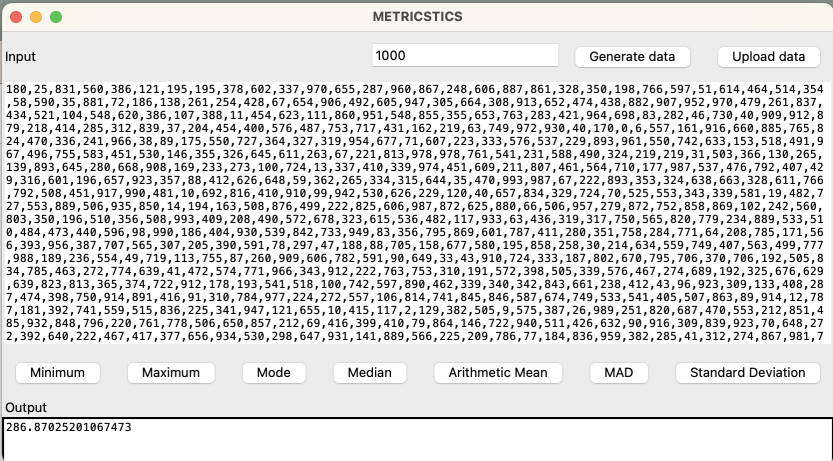
Test: Input 1	Input from input1_sequential.csv
Min	 <p>The screenshot shows the METRICSTICS application interface. The 'Input' field is set to 1000. The 'Generate data' and 'Upload data' buttons are visible. The 'Output' section displays the result of the 'Minimum' calculation, which is 1. The list of generated data is visible in the background.</p>
Max	 <p>The screenshot shows the METRICSTICS application interface. The 'Input' field is set to 1000. The 'Generate data' and 'Upload data' buttons are visible. The 'Output' section displays the result of the 'Maximum' calculation, which is 1000. The list of generated data is visible in the background.</p>

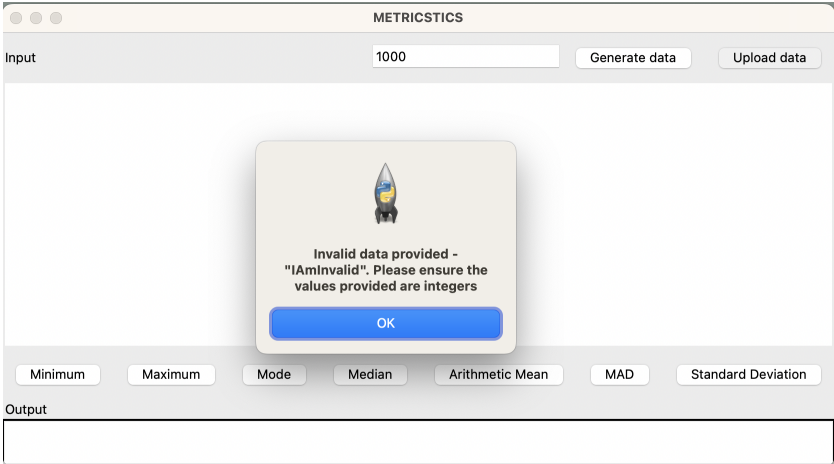
[illegible]

Standard Deviation	<div><div><div><div><div></div><div></div><div></div></div><div>METRICSTICS</div></div><div><div>Input</div><div>1000</div><div>Generate data</div><div>Upload data</div></div><div>1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612,</div><div><div>Minimum</div><div>Maximum</div><div>Mode</div><div>Median</div><div>Arithmetic Mean</div><div>MAD</div><div>Standard Deviation</div></div><div>Output</div><div>289.1526284521236</div></div></div>
--------------------	--

Test: Random 1	Input from clicking on 'Generate Data' (Generated values are saved into 'random1.csv')
Min	<div><div><div><div><div></div><div></div><div></div></div><div>METRICSTICS</div></div><div><div>Input</div><div>1000</div><div>Generate data</div><div>Upload data</div></div><div>180, 25, 831, 560, 386, 121, 195, 195, 378, 602, 337, 970, 655, 287, 960, 867, 248, 606, 887, 861, 328, 350, 198, 766, 597, 51, 614, 464, 514, 354, 58, 590, 35, 881, 72, 186, 138, 261, 254, 428, 67, 654, 906, 492, 605, 947, 305, 664, 308, 913, 652, 474, 438, 882, 907, 952, 970, 479, 261, 837, 434, 521, 104, 548, 620, 386, 107, 388, 11, 454, 623, 111, 860, 951, 548, 855, 355, 653, 763, 283, 421, 964, 698, 83, 282, 46, 730, 40, 909, 912, 8, 79, 218, 414, 285, 312, 839, 37, 204, 454, 400, 576, 487, 753, 717, 431, 162, 219, 63, 749, 972, 930, 40, 170, 0, 6, 557, 161, 916, 660, 885, 765, 8, 24, 470, 336, 241, 966, 38, 89, 175, 550, 727, 364, 327, 319, 954, 677, 71, 607, 223, 333, 576, 537, 229, 893, 961, 550, 742, 633, 153, 518, 491, 9, 67, 496, 755, 583, 451, 530, 146, 355, 326, 645, 611, 263, 67, 221, 813, 978, 978, 761, 541, 231, 588, 490, 324, 219, 219, 31, 503, 366, 130, 265, 139, 893, 645, 280, 668, 908, 169, 233, 273, 100, 724, 13, 337, 410, 339, 974, 451, 609, 211, 807, 461, 564, 710, 177, 987, 537, 476, 792, 407, 42, 9, 316, 601, 196, 657, 923, 357, 806, 412, 626, 648, 59, 362, 265, 334, 315, 644, 35, 470, 993, 987, 67, 222, 893, 353, 324, 638, 663, 328, 611, 766, 792, 508, 451, 917, 990, 481, 10, 692, 816, 410, 910, 99, 942, 530, 626, 229, 120, 40, 657, 834, 329, 724, 70, 525, 553, 343, 339, 581, 19, 482, 7, 27, 553, 889, 506, 935, 850, 14, 194, 163, 508, 876, 499, 222, 825, 606, 987, 872, 625, 880, 66, 506, 957, 279, 872, 752, 858, 869, 102, 242, 560, 803, 350, 196, 510, 356, 508, 993, 409, 208, 490, 572, 678, 323, 615, 536, 482, 117, 933, 63, 436, 319, 317, 750, 565, 820, 779, 234, 889, 533, 51, 0, 484, 473, 440, 596, 98, 990, 186, 404, 930, 539, 842, 733, 949, 83, 356, 795, 869, 601, 787, 411, 280, 351, 758, 284, 771, 64, 208, 785, 171, 56, 6, 393, 956, 387, 707, 565, 307, 205, 390, 591, 78, 297, 47, 188, 88, 705, 158, 677, 580, 195, 858, 258, 30, 214, 634, 559, 749, 407, 563, 499, 777, 988, 189, 236, 554, 49, 719, 113, 755, 87, 260, 909, 606, 782, 591, 90, 649, 33, 43, 910, 724, 333, 187, 802, 670, 795, 706, 370, 706, 192, 505, 8, 34, 785, 463, 272, 774, 639, 41, 472, 574, 771, 966, 343, 912, 222, 763, 753, 310, 191, 572, 398, 505, 339, 576, 467, 274, 689, 192, 325, 676, 629, 639, 823, 813, 365, 374, 722, 912, 178, 193, 541, 518, 100, 742, 597, 890, 462, 339, 340, 342, 843, 661, 238, 412, 43, 96, 923, 309, 133, 408, 28, 7, 474, 398, 750, 914, 891, 416, 91, 310, 784, 977, 224, 272, 557, 106, 814, 741, 845, 846, 587, 674, 749, 533, 541, 405, 507, 863, 89, 914, 12, 78, 7, 181, 392, 741, 559, 515, 836, 225, 341, 947, 121, 655, 10, 415, 117, 2, 129, 382, 505, 9, 575, 387, 26, 989, 251, 820, 687, 470, 553, 212, 851, 4, 85, 932, 848, 796, 220, 761, 778, 506, 650, 857, 212, 69, 416, 399, 410, 79, 864, 146, 722, 940, 511, 426, 632, 90, 916, 309, 839, 923, 70, 648, 27, 2, 392, 640, 222, 467, 417, 377, 656, 934, 530, 298, 647, 931, 141, 889, 566, 225, 209, 786, 77, 184, 836, 959, 382, 285, 41, 312, 274, 867, 981, 7</div><div><div>Minimum</div><div>Maximum</div><div>Mode</div><div>Median</div><div>Arithmetic Mean</div><div>MAD</div><div>Standard Deviation</div></div><div>Output</div><div>0</div></div></div>
Max	<div><div><div><div><div></div><div></div><div></div></div><div>METRICSTICS</div></div><div><div>Input</div><div>1000</div><div>Generate data</div><div>Upload data</div></div><div>180, 25, 831, 560, 386, 121, 195, 195, 378, 602, 337, 970, 655, 287, 960, 867, 248, 606, 887, 861, 328, 350, 198, 766, 597, 51, 614, 464, 514, 354, 58, 590, 35, 881, 72, 186, 138, 261, 254, 428, 67, 654, 906, 492, 605, 947, 305, 664, 308, 913, 652, 474, 438, 882, 907, 952, 970, 479, 261, 837, 434, 521, 104, 548, 620, 386, 107, 388, 11, 454, 623, 111, 860, 951, 548, 855, 355, 653, 763, 283, 421, 964, 698, 83, 282, 46, 730, 40, 909, 912, 8, 79, 218, 414, 285, 312, 839, 37, 204, 454, 400, 576, 487, 753, 717, 431, 162, 219, 63, 749, 972, 930, 40, 170, 0, 6, 557, 161, 916, 660, 885, 765, 8, 24, 470, 336, 241, 966, 38, 89, 175, 550, 727, 364, 327, 319, 954, 677, 71, 607, 223, 333, 576, 537, 229, 893, 961, 550, 742, 633, 153, 518, 491, 9, 67, 496, 755, 583, 451, 530, 146, 355, 326, 645, 611, 263, 67, 221, 813, 978, 978, 761, 541, 231, 588, 490, 324, 219, 219, 31, 503, 366, 130, 265, 139, 893, 645, 280, 668, 908, 169, 233, 273, 100, 724, 13, 337, 410, 339, 974, 451, 609, 211, 807, 461, 564, 710, 177, 987, 537, 476, 792, 407, 42, 9, 316, 601, 196, 657, 923, 357, 806, 412, 626, 648, 59, 362, 265, 334, 315, 644, 35, 470, 993, 987, 67, 222, 893, 353, 324, 638, 663, 328, 611, 766, 792, 508, 451, 917, 990, 481, 10, 692, 816, 410, 910, 99, 942, 530, 626, 229, 120, 40, 657, 834, 329, 724, 70, 525, 553, 343, 339, 581, 19, 482, 7, 27, 553, 889, 506, 935, 850, 14, 194, 163, 508, 876, 499, 222, 825, 606, 987, 872, 625, 880, 66, 506, 957, 279, 872, 752, 858, 869, 102, 242, 560, 803, 350, 196, 510, 356, 508, 993, 409, 208, 490, 572, 678, 323, 615, 536, 482, 117, 933, 63, 436, 319, 317, 750, 565, 820, 779, 234, 889, 533, 51, 0, 484, 473, 440, 596, 98, 990, 186, 404, 930, 539, 842, 733, 949, 83, 356, 795, 869, 601, 787, 411, 280, 351, 758, 284, 771, 64, 208, 785, 171, 56, 6, 393, 956, 387, 707, 565, 307, 205, 390, 591, 78, 297, 47, 188, 88, 705, 158, 677, 580, 195, 858, 258, 30, 214, 634, 559, 749, 407, 563, 499, 777, 988, 189, 236, 554, 49, 719, 113, 755, 87, 260, 909, 606, 782, 591, 90, 649, 33, 43, 910, 724, 333, 187, 802, 670, 795, 706, 370, 706, 192, 505, 8, 34, 785, 463, 272, 774, 639, 41, 472, 574, 771, 966, 343, 912, 222, 763, 753, 310, 191, 572, 398, 505, 339, 576, 467, 274, 689, 192, 325, 676, 629, 639, 823, 813, 365, 374, 722, 912, 178, 193, 541, 518, 100, 742, 597, 890, 462, 339, 340, 342, 843, 661, 238, 412, 43, 96, 923, 309, 133, 408, 28, 7, 474, 398, 750, 914, 891, 416, 91, 310, 784, 977, 224, 272, 557, 106, 814, 741, 845, 846, 587, 674, 749, 533, 541, 405, 507, 863, 89, 914, 12, 78, 7, 181, 392, 741, 559, 515, 836, 225, 341, 947, 121, 655, 10, 415, 117, 2, 129, 382, 505, 9, 575, 387, 26, 989, 251, 820, 687, 470, 553, 212, 851, 4, 85, 932, 848, 796, 220, 761, 778, 506, 650, 857, 212, 69, 416, 399, 410, 79, 864, 146, 722, 940, 511, 426, 632, 90, 916, 309, 839, 923, 70, 648, 27, 2, 392, 640, 222, 467, 417, 377, 656, 934, 530, 298, 647, 931, 141, 889, 566, 225, 209, 786, 77, 184, 836, 959, 382, 285, 41, 312, 274, 867, 981, 7</div><div><div>Minimum</div><div>Maximum</div><div>Mode</div><div>Median</div><div>Arithmetic Mean</div><div>MAD</div><div>Standard Deviation</div></div><div>Output</div><div>1000</div></div></div>

Mode	<p>The screenshot shows a web application titled "METRICSTICS" with an "Input" field containing "1000". There are buttons for "Generate data" and "Upload data". Below the input field, a large list of 1000 numbers is displayed. At the bottom, there are buttons for "Minimum", "Maximum", "Mode", "Median", "Arithmetic Mean", "MAD", and "Standard Deviation". The "Output" field shows "0: 5 count(s)".</p>
Median	<p>The screenshot shows the same web application as above, but the "Output" field now displays "630.5".</p>
MAD	<p>The screenshot shows the same web application as above, but the "Output" field now displays "244.45860400000004".</p>

Standard Deviation	
--------------------	--

Test: Input 3	Input from input3_error.csv
Error input	

1.3 Problem 5: Cyclomatic Number

1.3.1 Calculating the Cyclomatic Number (Cyclomatic Complexity)

To implement METRICSTICS, the object-oriented paradigm was used. There are 4 classes that were defined, with each assuming responsibility for specific functionalities:

1. **main.py**: involves the handling of the GUI functionalities
2. **calculator.py**: descriptive statistics calculation functions
3. **csv_processor.py**: csv processing and functions
4. **data_validator.py**: functions to transform and validate data input for descriptive statistics calculation

To obtain the cyclomatic complexity for each class, a tool called Radon was used. Radon is a Python tool that computes various code metrics, including cyclomatic complexity, which is a quantitative measure of code paths and a predictor of its maintainability. Utilizing Radon, we analyzed the METRICSTICS system to assess its complexity and infer the quality of the codebase. This report presents the cyclomatic complexity for each module and discusses the implications of these metrics on the system's overall maintainability and robustness. The following sections detail the complexity scores and provide a qualitative interpretation relative to standard thresholds.

1. Cyclomatic Complexity for main.py

```
[jothibasulkv@jothis-mbp code % radon cc main.py -a
main.py
  F 10:0 browse_csv - A
  F 32:0 generate - A
  F 42:0 get_generate_input_text - A
  F 21:0 calculate_input - A
  F 53:0 generate_random - A

5 blocks (classes, functions, methods) analyzed.
Average complexity: A (3.0)
[jothibasulkv@jothis-mbp code % radon cc main.py -s
main.py
  F 10:0 browse_csv - A (4)
  F 32:0 generate - A (4)
  F 42:0 get_generate_input_text - A (3)
  F 21:0 calculate_input - A (2)
  F 53:0 generate_random - A (2)
```

Figure 1.4: Cyclomatic Complexity for main.py

Average Cyclomatic Complexity: A (3.0)

2. Cyclomatic Complexity for calculator.py

```
jothibasulkv@jothis-mbp code % radon cc calculator.py -a
calculator.py
C 1:0 Calculator - A
M 11:2 Calculator.is_data_exists - A
M 16:2 Calculator.get_min - A
M 25:2 Calculator.get_max - A
M 34:2 Calculator.get_mode - A
M 47:2 Calculator.get_median - A
M 56:2 Calculator.get_arithmetic_mean - A
M 64:2 Calculator.get_mean_abs_deviation - A
M 72:2 Calculator.get_standard_deviation - A
M 80:2 Calculator.sqrt - A
M 2:2 Calculator.__init__ - A
M 5:2 Calculator.set_data - A
M 8:2 Calculator.clear_data - A

13 blocks (classes, functions, methods) analyzed.
Average complexity: A (2.1538461538461537)
jothibasulkv@jothis-mbp code % radon cc calculator.py -s
calculator.py
C 1:0 Calculator - A (3)
M 11:2 Calculator.is_data_exists - A (3)
M 16:2 Calculator.get_min - A (3)
M 25:2 Calculator.get_max - A (3)
M 34:2 Calculator.get_mode - A (3)
M 47:2 Calculator.get_median - A (2)
M 56:2 Calculator.get_arithmetic_mean - A (2)
M 64:2 Calculator.get_mean_abs_deviation - A (2)
M 72:2 Calculator.get_standard_deviation - A (2)
M 80:2 Calculator.sqrt - A (2)
M 2:2 Calculator.__init__ - A (1)
M 5:2 Calculator.set_data - A (1)
M 8:2 Calculator.clear_data - A (1)
```

Figure 1.5: Cyclomatic Complexity for calculator.py

Average Cyclomatic Complexity: A (2.1538461538461537)

3. Cyclomatic Complexity for csv_processor.py

```
jothibasulkv@jothis-mbp code % radon cc csv_processor.py -a
csv_processor.py
C 3:0 CsvProcessor - A
M 4:2 CsvProcessor.__init__ - A
M 7:2 CsvProcessor.read_csv_and_validate - A

3 blocks (classes, functions, methods) analyzed.
Average complexity: A (1.3333333333333333)
jothibasulkv@jothis-mbp code % radon cc csv_processor.py -s
csv_processor.py
C 3:0 CsvProcessor - A (2)
M 4:2 CsvProcessor.__init__ - A (1)
M 7:2 CsvProcessor.read_csv_and_validate - A (1)
```

Figure 1.6: Cyclomatic Complexity for csv_processor.py

Average Cyclomatic Complexity: A (1.3333333333333333)

4. Cyclomatic Complexity for data_validator.py

```
jothibasulkv@jothis-mbp code % radon cc data_validator.py -a
data_validator.py
M 5:2 DataValidator.process - A
C 1:0 DataValidator - A
M 10:2 DataValidator.validate - A
M 2:2 DataValidator.__init__ - A

4 blocks (classes, functions, methods) analyzed.
Average complexity: A (2.5)
jothibasulkv@jothis-mbp code % radon cc data_validator.py -s
data_validator.py
M 5:2 DataValidator.process - A (4)
C 1:0 DataValidator - A (3)
M 10:2 DataValidator.validate - A (2)
M 2:2 DataValidator.__init__ - A (1)
```

Figure 1.7: Cyclomatic Complexity for data_validator.py

Average Cyclomatic Complexity: A (2.5)

1.3.2 Qualitative Conclusions

1. Qualitative Conclusions for main.py

- (a) The functions `browse_csv` and `generate` have the highest complexity scores of 4, which indicates that they have multiple paths for execution but remain within a reasonable range for understanding and maintenance.
- (b) The `get_generate_input_text` function has a slightly lower complexity score of 3, suggesting that it is simpler than the aforementioned functions, likely due to fewer conditional paths.
- (c) The `calculate_input` and `generate_random` functions have the lowest complexity scores of 2, which are indicative of straightforward, linear execution paths with minimal branching.

2. Qualitative Conclusions for calculator.py

- (a) The methods generally have low complexity scores (1 to 3), indicating that each method is likely doing one specific task with a clear and straightforward logic flow.
- (b) The uniformity of scores (mostly 3s and 2s) suggests consistent coding practices and potentially similar levels of abstraction across different methods.
- (c) An average complexity score of slightly over 2 suggests that the `calculator.py` module is well within the range of good maintainability and should be straightforward to test.
- (d) Such a low average score is a positive indicator of code quality, implying that there are few conditional branches within each method, which aligns with best practices in writing clear and concise code.

3. Qualitative Conclusions for csv_processor.py

- (a) All methods in the `csv_processor.py` module have low complexity scores, with the `CsvProcessor` class itself scoring a 2 and the methods `__init__` and `read_csv_and_validate` both scoring a 1.
- (b) This indicates that the methods have a linear flow with minimal branching, which translates to a straightforward logic that is easy to follow and maintain.
- (c) The average complexity of approximately 1.33 suggests that the module is very well-designed, with simplicity at its core.

- (d) The module's low complexity is indicative of a high-quality design that should facilitate easy testing and low risk of defects.

4. Qualitative Conclusions for `data_validator.py`

- (a) `DataValidator` (the class itself) has a complexity of 3, suggesting that the class as a whole has some conditional logic but is not overly complex.
- (b) The `__init__` function has the lowest complexity, which is expected as constructors typically have a clear and single path of execution.
- (c) The function `process` has the highest complexity with a score of 4, which might indicate multiple paths or decision points within the method, yet it remains at an acceptable level of complexity.
- (d) The `validate` function has a complexity of 2, which is indicative of a straightforward method with one or two decision points or loops.

1.4 Problem 6: Object-oriented Metrics

1.4.1 Calculating the Object-Oriented Metrics - WMC, CF & LCOM

1. Weighted Method Count (WMC)

The weighted method count (WMC) is calculated by assigning complexity values (A-values) to each method based on their cyclomatic complexity:

(a) Calculator Class

- i. Calculator.is_data_exists - A(3)
- ii. Calculator.get_min - A(3)
- iii. Calculator.get_max - A(3)
- iv. Calculator.get_mode - A(3)
- v. Calculator.get_median - A(2)
- vi. Calculator.get_arithmetic_mean - A(2)
- vii. Calculator.get_mean_abs_deviation - A(2)
- viii. Calculator.get_standard_deviation - A(2)
- ix. Calculator.sqrt - A(2)
- x. Calculator.__init__ - A(1)
- xi. Calculator.set_data - A(1)
- xii. Calculator.clear_data - A(1)

$$WMC = 3 + 3 + 3 + 3 + 2 + 2 + 2 + 2 + 2 + 1 + 1 + 1 \quad (1.20)$$

$$= 22 \quad (1.21)$$

Therefore, the WMC of the **Calculator** class is 22.

(b) CsvProcessor Class

- i. CsvProcessor.__init__ - A(1)
- ii. CsvProcessor.read_csv_and_validate - A(1)

$$WMC = 1 + 1 \quad (1.22)$$

$$= 2 \quad (1.23)$$

Therefore, the WMC of the **CsvProcessor** class is 2.

(c) DataValidator Class

- i. DataValidator.process - A(4)
- ii. DataValidator.validate - A(2)
- iii. DataValidator.__init__ - A(1)

$$WMC = 4 + 2 + 1 \quad (1.24)$$

$$= 7 \quad (1.25)$$

Therefore, the WMC of the **DataValidator** class is 7.

2. Coupling Factor (CF) Relationships

- (a) `IsClient(Calculator, CsvProcessor):`
Established when creating a `CsvProcessor` instance in `browse_csv()`
- (b) `IsClient(Calculator, DataValidator):`
Established when creating a `DataValidator` instance in `browse_csv()`
- (c) `IsClient(CsvProcessor, Calculator):`
Established when calling `CsvProcessor.read_csv_and_validate()`
- (d) `IsClient(CsvProcessor, DataValidator):`
Established when creating a `DataValidator` instance in `read_csv_and_validate()`
- (e) `IsClient(DataValidator, Calculator):`
Established when calling `DataValidator.process(input_text.get("1.0", tk.END))` in `calculate_input()`.
- (f) `IsClient(DataValidator, CsvProcessor):`
Established when calling `DataValidator.process(content)` in `read_csv_and_validate()`.
- (g) `IsClient(DataValidator, CsvProcessor):`
Additional relationship when calling `DataValidator.process(content)` in `read_csv_and_validate()`

$$CF = \frac{1 + 1 + 1 + 1 + 1 + 1 + 1}{3^2 - 3} \quad (1.26)$$

$$= \frac{7}{6} \quad (1.27)$$

3. Lack of Cohesion of Methods (LCOM)

(a) Calculator Class

i. Methods:

- `__init__`: Initializes the `self.data` attribute
- `set_data`: Sets the value of `self.data`
- `clear_data`: Sets `self.data` to `None`
- `is_data_exists`: Checks if `self.data` is not `None` and not empty
- `get_min`, `get_max`, `get_mode`, `get_median`, `get_arithmetic_mean`, `get_mean_abs_deviation`, `get_standard_deviation`: All these methods access `self.data` to perform statistical calculations
- `sqrt`: Uses babylonian method to calculate the square root (approximates a hypotenuse)

ii. Analysis:

- Attributes: `self.data`
- Total Methods (m): 9
- Number of Attributes (a): 1

$$LCOM = \frac{1}{1 * 9 - 9(1 - 9)} \quad (1.28)$$

$$= \frac{1}{9 + 72} \quad (1.29)$$

$$= \frac{1}{81} \quad (1.30)$$

$$\approx 0.0123 \quad (1.31)$$

(b) CsvProcessor Class

- i. Methods:
 - `__init__`: Initializes the `self.file` attribute
 - `read_csv_and_validate`: Reads the content of the file, creates an instance of `DataValidator`, and then calls the `process` method on that instance
- ii. Analysis:
 - Attributes: `self.file`, instance of `DataValidator`
 - Total Methods (m): 2
 - Number of Attributes (a): 2

$$LCOM = \frac{1}{2 * 2 - 2(1 - 2)} \quad (1.32)$$

$$= \frac{1}{6} \quad (1.33)$$

$$\approx 0.1667 \quad (1.34)$$

(c) `DataValidator` Class

- i. Methods:
 - `__init__`: Initializes the instance of the class
 - `process`: Takes a string `str_data` as input, checks if it's not `None` and not empty, and then calls the `validate` method for each value obtained by splitting `str_data`
 - `validate`: Takes a value as input, strips whitespace, checks if it's numeric, and raises an exception if not. Returns the integer value.
- ii. Analysis:
 - Attributes: `self.data`
 - Total Methods (m): 3
 - Number of Attributes (a): 1

$$LCOM = \frac{1}{1 * 3 - 3(1 - 3)} \quad (1.35)$$

$$= \frac{1}{9} \quad (1.36)$$

$$\approx 0.1111 \quad (1.37)$$

1.4.2 Qualitative Conclusions

1. Weighted Method Count (WMC)

- (a) `Calculator` Class (WMC = 22):
 - i. The complexity of the `Calculator` class is indicated by its comparatively high WMC.
 - ii. Various methods, such `get_min` and `get_max`, add to the total complexity.
- (b) `CsvProcessor` Class (WMC = 2):
 - i. The low WMC of the `CsvProcessor` class indicates a lesser level of complexity.
 - ii. This class appears to have fewer methods and simpler functionality.
- (c) `DataValidator` Class (WMC = 7):
 - i. The moderate WMC of the `DataValidator` class indicates moderate complexity.
 - ii. A major factor in the total complexity of the class is the `process` method.

2. Coupling Factor (CF)

The object-oriented design (OOD) has a moderate level of coupling between classes, as shown by the computed Coupling Factor (CF) of roughly 1.17. The research highlights how classes interact in a balanced way, with a focus on flexibility and maintainability. The new relationship, which suggests well-distributed interactions without excessive reliance, confirms the modest coupling of the design. This complies with sound design guidelines and makes the system more flexible and cohesive.

3. Lack of Cohesion of Methods (LCOM*)

The coherence of practises and characteristics within each class is revealed by the LCOM5 scores. Higher cohesiveness is indicated by lower LCOM5 values, which imply closer relationships between methods inside a class. The following are the outcomes:

- (a) Calculator Class: LCOM* with a range of 0.0123 (Weak cohesiveness; approaches are tangentially connected)
- (b) CsvProcessor Class: LCOM* approximately 0.1667 (Moderate cohesiveness; moderate relationship between approaches)
- (c) DataValidator Class: LCOM* approximately 0.1111 (Moderate cohesiveness; methods are moderately connected)

These numbers imply that the cohesion of the classes is generally moderate, with Calculator having the lowest cohesion.

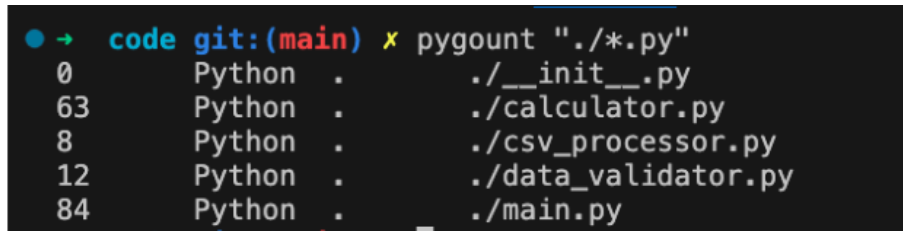
1.5 Problem 7: Physical and Logical SLOC

1.5.1 Calculating the Physical SLOC and Logical SLOC

1. Physical SLOC

This project uses the `pygount` tool to calculate the Physical SLOC for METRICSTICS. It is a command line tool to count the physical lines of code, similar to tools like `sloccount` and `cloc`, but instead uses the `pygments` package for source code parsing and can analyze any programming language that `pygments` support.[2]

There are several operations that are applicable to certain languages which are treated as white space when detected as they are considered “no operations”. For example Python’s `pass` or Transact-SQL’s `begin` and `end`.



```
● → code git:(main) x pygount "./*.py"
0      Python .      ./__init__.py
63     Python .      ./calculator.py
8      Python .      ./csv_processor.py
12     Python .      ./data_validator.py
84     Python .      ./main.py
```

Figure 1.8: Physical SLOC calculation using `pygount` tool

2. Logical SLOC

The logical SLOC of the METRICSTICS app is calculated manually based on the number of logical statements or instructions in the source code that affect the program’s control flow and/or behaviour. Only the lines of code that contain actual executable operations were counted. This includes conditional statements, loops, actual operational code within functions/methods, such as data processing, calculations, exception handling, input/output operations, and other executed operations

Class Name	Physical SLOC	Logical SLOC
calculator.py	63	38
csv_processor.py	8	3
data_validator.py	11	6
main.py	84	59
Total:	167	106

1.5.2 Qualitative Conclusions

1. A system with a total of 167 physical SLOC implies a smaller size, indicating lower complexity. The reduced size suggests easier maintenance due to its compact nature.
2. An 106 count of logical Source Lines of Code (SLOC) implies that the software project is relatively small compared to generic systems, potentially featuring a more straightforward and manageable source code. However, it’s important to note that relying solely on logical SLOC may not present a complete picture. To gain a more comprehensive understanding of the project, it is crucial to take into account other metrics and factors as well.
3. The difference between the logical SLOC of 106 and the physical SLOC of 167 is significant and may require further review of the code to confirm whether there are excessive comments, structural elements, or the code can be further optimized to increase the competition of the code. strength and maintenance.

1.6 Problem 8: Establishing Relation between Logical SLOC and WMC

1.6.1 Scatter Plot

Class Name	Logical SLOC	WMC
Calculator	38	22
CsvProcessor	3	2
DataValidator	6	7

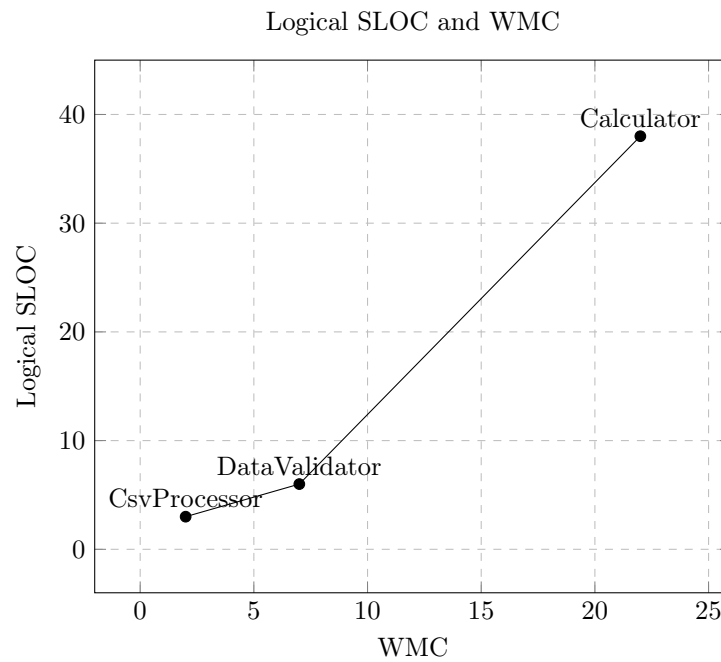


Figure 1.9: Scatter Plot (Logical SLOC and WMC)

Figure 1.9 shows the scatter plot between the data for Logical SLOC and WMC obtained from METRICSTICS. It can be noted that as the WMC increases, the logical SLOC increases, thus it can be said that the logical SLOC is directly proportional to the WMC.

1.6.2 Correlation Coefficient

Since the values of x 's and y 's are non-normally distributed as per the scatter plot shown above, the Spearman's Rank Correlation Coefficient (r_s) can be used to find the correlation coefficient as it is a measure of association for attributes values that are not distributed normally.

WMC(x_i)	Rank(x_i)	SLOC(y_i)	Rank(y_i)	d	d ²
22	3	38	3	0	0
2	1	3	1	0	0
7	2	6	2	0	0

Table 1.6: Summary of statistics for calculating the r_s

$$r_s = 1 - \frac{6 \sum_{i=1}^3 d^2}{3^3 - 3} \quad (1.38)$$

$$= 1 - \frac{6 * 0}{3^3 - 3} \quad (1.39)$$

$$= 1 - 0 \quad (1.40)$$

$$= 1 \quad (1.41)$$

The value of $r_s = 1$ indicate a strong positive correlation.

Bibliography

- [1] About python. <https://www.python.org/about/>. Accessed: 2023-11-18.
- [2] pygount. <https://pygount.readthedocs.io/en/latest/index.html>. Accessed: 2023-11-19.
- [3] Tkinter wiki. <https://wiki.python.org/moin/TkInter>. Accessed: 2023-11-18.