# Supplementary File 4

# Joint-view trustworthiness (JVT) pseudocode and computational complexity analysis

*Pseudo-code of JVT*

, – set of names of the datasets to load. Each is composed of 4 variables: the corresponding , , , .

– the actual dataset within each , where features/genes are on rows and samples are on columns. Each is organized having the same genes and in the same order on the rows

– an array of gene names common to each dataset

– a 2D array where each row corresponds to in each reporting for each sample the name of one of the two groups to which it belongs in

– an array containing the name of one of the two groups of samples that should be used as positive class for AUC-ROC evaluation; each corresponds to each

– an array containing the names of genes used for computing the combinatorial marker

– a numerical array of the same length as containing one of the values in { for each gene, indicating whether alignment is needed or not

– type of gene normalization involved. Offered options: ‘*log10*’ and ‘*z-score*’

– an integer indicating the number of times the samples are drawn to build null model distributions

– length of

– length of

– size of

– number of samples of a

– matrix, where rows 1 and 2 store minimum AUC score and p-value of minimum AUC score, respectively, as computed by JVT. Each column corresponds to each marker (single+combinatory).

|  |
| --- |
| **Alg1. Joint Multiview Trustworthiness** |

1. datasets , a Combinatorial Marker array , an array , normalization type , sampling times

2. find gene indices, , in via any using its and //

// *initialisation of variables begins*

3. generate a series // *used as indices to datasets* ;

4.//  *contains with removed* ;

5.

6. append back to //

7. **if** **OR** **then** // *for randomisation; if , no need to randomise*

8. uniformly generate random integers //

9. //  *is updated and now contains random indices from and has length*

10. append back to //

11. **end if**

12. //

13. initialise zero matrix, , and zero matrix,

//  *and are used to store single and combinatory gene AUC performance,* // *respectively, and have the same number of columns as the no. of datasets*

//  *for matrix and for matrix*

// *initialisation of variables ends*

14. **for** **do** // *for each dataset* ;

15. load dataset // *this loads all the variables, , ,* // *and corresponding to dataset,*

16. perform *log* or *z-score* normalisation to corresponding to based on   
 // *if does not contain offered options, use log normalisation as default;*

17. **for**  **do** // *for all sampling times and*

// **≈** )

18.pick the index in and select the corresponding gene, in

19.

// *compute of in using the corresponding*   
// *and*  ;

20.  // *for single gene* ; *row, column*

21. **if** **then** // *for combinatory gene*

22.  **if**  **then**

23.  uniformly generate random integers //

24.

//  *contains random indices from and has length*

25. **elseif**  **then**

26. // *no randomisation; only marker indices used*

27. **end if**

28.  genes identified by indices in //

29. **for** **to** **do** // *for gene alignment ;*  ***≈***

30.  **if**  **then**

31. //

// *take the mean of the gene identified by its index*

32. // *align and update gene* //

33. **end if**

34.  **end for**

35. //*take the column-wise mean of genes* ;

36.

// *compute of using the corresponding and* // ;

37.

38. **end if**

39. **end for**

40. **end for**

41. compute the minimum of along each row //

// is a vector containing the minimum AUC performance of single gene  
 // across the datasets

42. compute minimum of along each row //

// is a vector containing the minimum AUC performance of combinatorial   
 // gene across the datasets

43. initialise zero matrix, // *for result compilation* ;

44. **for** **to** **do** // *for single marker* ;

45. value corresponding to the index //

46. (number of times // *p-value* ;

47. **end for**

48. // *for combinatory marker*

49. number of times

// *p-value* ;

50.append *‘Comb Marker’* to

51. set row names of {‘min AUC-ROC’, ‘JVT p-value’}

52. set column names of

53.

*Computational complexity of JVT*

We begin with the breakdown of computational complexity of JVT as presented in *Alg1* and then provide the overall associated complexity of JVT. From line 2 in *Alg1*, we can see the complexity of finding number of marker genes in a set of number of common genes is . Similarly, lines 3, 4 and 6 have the complexity of and , respectively. Line 8, where we generate random numbers and has the complexity of , where represents the sampling times, and line 9 runs at . In line no. 13, we create two zero matrices for storing the results, and , having the computational complexity of and , respectively, where is the number of datasets involved. The loop in line 14 has a total complexity of . This complexity can be arrived using lines 15 through 40, which compose the statements of the loop, as follows. Line no. 16, which normalizes each value in a given dataset has a complexity of . Line 17, which is also a loop, has a total complexity of . This can be arrived at by inspecting lines 19, 23, 28, 31, 32, 35 and 36. While, the complexities of these lines are easy to deduce and are given in the algorithm itself, lines 19, 29 and 36 are of particular interest here. Since line 19 computes the score of the gene with respect to the number of samples the gene carries, , it has the computational complexity of . The same can be said of line 36. Line 19 is a loop and performs the gene alignment with a complexity of . Putting all this together, we get the computational complexity of line 14 stated above. Furthermore, line 41, which computes the minimum of all the sampled genes, , across all the datasets, , has the complexity of . Similarly, for line 42, we have . Line 43 creates a zero matrix and intuitively runs at , and loop in line 44 at . Finally, line 49 has the computational complexity of . Summing up the algorithm’s complexity and simplifying, we get,

.

Clearly, if the values , and are highly dominated by the and , which is the case in our study, the effect of , and in the overall complexity of the JVT becomes negligible. Hence, the complexity boils down to , i.e., the JVT’s complexity is non-linear in and .