Similarity Ensemble Approach

Christian Micklisch, Manoj Krishna Mohan, AJ Daodu, Aniruddha Sudhindra Shirahatti

Abstract:

Similarity Ensemble Approach (SEA) is widely known for finding set similarities with a statistical approach to predict chemical similarities and side effects in drug-related studies. They help in studies related to screening of large databases containing structures of available (or potentially available) chemicals. This is based on the studies of similar property principle of Johnson and Maggiora, which states: *Similar compounds have similar properties*[1]. In this project, we developed a simulation of finding the similarity amongst sets of molecular topology. We propose a statistical method which creates bins that provide ranks for their similarity scores, i.e the closest sets are mapped using the Minimum Spanning Tree approach. The proposed approach includes a Java implementation that decreases the runtime from the previous Python implementation. We have run the simulations for different sample sizes from 100 to 1000. There is a significant increase in the runtime, in comparison with the existing python implementation.

1. Introduction:

The SEA method is motivated by two ideas, proteins can be related by the drug pharmacology and the network relationships can be used for discovering small molecules to the targets. It basically compares a set of ligands based on their topology. A sequence is generated for the molecule/ligand fingerprints for the topology that is represented, and from this sequence, we calculate the Raw Scores, and then we compare them with the Expected values from the background distribution which will lead to the representation of the Chemical Similarity Sets.

The approach leverages the basic principle as stated[1] that *Similar molecules likely have similar properties* Thus, the targets of a molecule can be predicted by identifying proteins with known ligands that are highly similar to the query molecule. Moreover, the previous implementation was in Python which resulted in a loss of efficiency. Thus we took the initiative to bring down the runtime implementation of the code by using Java and rebuilding the application/libraries from Python using Java. Finally after calculating we make use of Kruskal's Algorithm to view the similarity sets with a MST. Even though we increase the sample size, we can observe that the runtime is significantly less than runtime of Python code.

2. Similarity Ensemble Approach (Methodology):

In our implementation of the Similarity Ensemble Approach, we simulate set similarity from Keiser and Hert's [2] methods of "Calculating the Parameters of the Reference Database",

"Calculating Set-Wise Similarity Ensembles", "Building a Similarity Network". We describe our implementations in section 2.1 to 2.3.

Each section allows us to calculate critical information to help build the Similarity Network. In Section 2.1 we determine the Chemical Similarity Set, Sample Factor Pairs C_a and C_b, best Time Step, t_i, and nonlinear fit for the mean plots and standard deviation plots that generate our y_micro and y_sigma functions respectively that are utilized by sections 2.2 and 2.3. Section 2.2 utilizes those generated values and calculate a list of optimal C_a values that are provided to build our similarity network in section 2.3.

2.1 Calculating the Reference Database:

In our implementation of the parameter calculation for the reference database we generate several thousand samples of numerical pairs and compare them to determine the optimal t_i, and nonlinear fitter functions, y_micro and y_sigma, along with defining the chemical similarity set.

In steps 1 - 3 we define our s_min and s_max values, generate our sample from our defined sample size and our range s_min^2 and s_max^2, and gather our factors from each of our samples determined. In step 4 we collect 30 pairs of factors at random from each samples respective factor list, we diverge from Keiser and Hert's [2] Note 7 where if our factor size is less than 30 than we only make selections from our factors provided. This random factor list then generates our two lists *a* and *b* in respect to our samples list. a consists of the random factor selected, *f_i*, while b consists of the sample over the random factor, *s_i*/*f_i*. In our implementation we determined that create a list of 30 sample factor pairs proved as effective at containing two separate lists of *a* and *b*, this implementation is contained in the `SampleFactorPair.java` file.

In step 5 we defined our Chemical Similarity Set as a two-dimensional array containing a row and column for each sample determined in step 2. Each row-column pair allows us to define a chemical similarity between each of the samples. In our simulation sample values for the row-column pair are a simple random value between zero and one. If the row and column equal each other the chemical similarity is one; the chemical similarity between the same chemical samples are one.

In step 6 we generate our timesteps between zero and one at increments of 0.01 and create a list of raw scores for each time step from the random factors in the sample factor pair list. In our implementation we check if the chemical similarity value is high enough to add to the raw score; we ignore sample factor pair lists that produce raw scores of zero. In step 7 we then take all of the raw score lists and generate a csv for each time step. This csv is converted to a plot where the x axis represents the product size of the sets *a* and *b* combined and the y axis represents the raw score.

We utilize the plots generated in step 7 to create the product size vs mean of raw scores plot in step 8. From this step we determine the equation for y_micro. Our method for determining the nonlinear fit equation is by utilizing apache.commons PolynomialCurveFitter [3] with a factor size of 10. This provided us with accurate curves generated in the scatter plots. In step 9 we bin the means found by their respective product size and calculate the standard deviation for the means found. This step provides us with the non linear fit for y_sigma.

Unlike Python, Java does not provide with utility libraries to calculate the standard deviation and the mean. So we wrote a Stats Class, which takes in an array of values and returns the mean and Standard Deviation of the same.

In step 10 we then calculate the z scores for each time step and plot the histogram of the zscores to their frequency in which they occur. We then utilize the z score plot to calculate the nonlinear fits for Gaussian and EVD distribution in step 11. We calculate a goodness fit by calculating the mean distance from the y values of the zscore histogram plot and the resulting nonlinear fit for Gaussuian[4] and EVD[5] distributions. The EVD distributions were determined by the scipy.stats gumbl_r.pdf function and converted to Java in the `Utils/Gumble.java` file. To ensure that we select the best fit that is closer to EVD than Gaussian we divide the EVD mean to the Gaussian mean. If the result is less than 1 then the fit is closer towards the EVD distribution, if it is greater than 1 then the fit is closer to the Gaussian distribution. We then select the time step with the lowest score to utilize with our set-wise similarity calculation and our network building.

2.2 Calculating the Set-Wise Similarity:

In our set-wise similarity calculation we are initially provided the optimal time step, chemical similarity set, sample factor pair list, y_micro, and y_sigma equations. The goal for the chemical set-wise similarity calculations are to provide a list of optimal samples that are chemically similar from the provided data. We do this by recalculating the raw and z scores to determine the e scores as defined by Keiser and Hert. The e score contains the p-value and the BLAST-like E-value.

In step 6 of Keiser and Hert compare each of the b_i values from C_b by their respective E-value where the best scores are those that are reaching zero. In our implementation we take the p-value and add it, if it's below a threshold of 0.5, to the list of best score samples, or optimal samples. We've found that the E-value's calculated is always greater than zero as the number of set-vs-set comparisons is large. This list of optimal samples from the e score is then passed to the similarity network build section.

2.3 Building the Similarity Network:

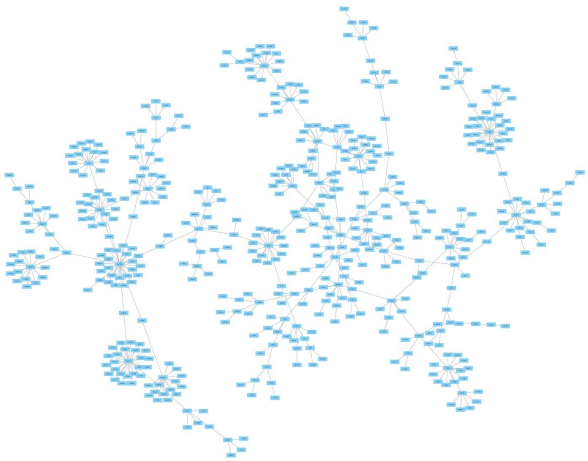
In our similarity network build process we are provided the list of optimal samples, the optimal time step, and the chemical similarity set. The similarity network is structured around the optimal samples provided where each sample is compared to one another to generate a two dimensional matrix of e scores that represent the escore of a row sample to a column sample. This matrix is then utilized to create a list of vertices and edges where each vertex is connected to all other vertice excluding itself. Kruskals is then run to generate the shortest path as described by Keiser and Hert.

Kruskal's MST Algorithm:

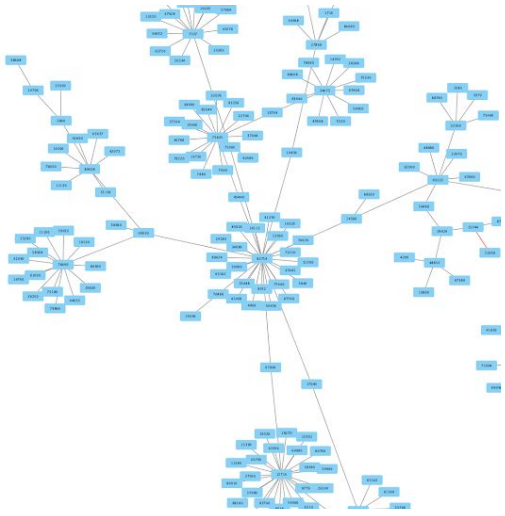
K-Means is one of the well known algorithms for clustering. But the Minimum Spanning Tree gives an advantage for clustering, because the algorithm is based on a trimmed Euclidean Minimum Spanning Tree. In MST-based clustering, the weight for each edge is considered as the Euclidean distance between the end points forming that edge.
The Kruskal's Algorithm[9] follows the following steps:
1. Sort the edges based on their weights in ascending order.
2. Pick the start, check if it forms a cycle with the spanning tree formed. If there is no cycle, we can include this edge. Else, we can discard it.
3. Step 2 is repeated until all the edges except for the one's which form a cycle are discarded.

3. Results

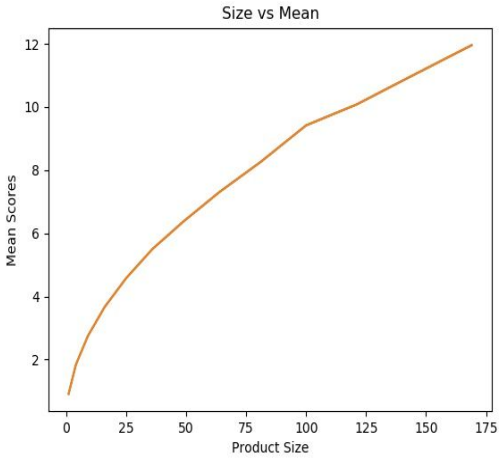
Set Similarity for a sample size of 35000, and s_min of 10 and s_max of 300.

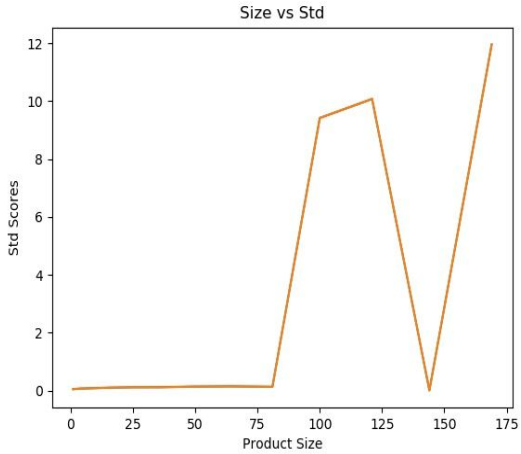


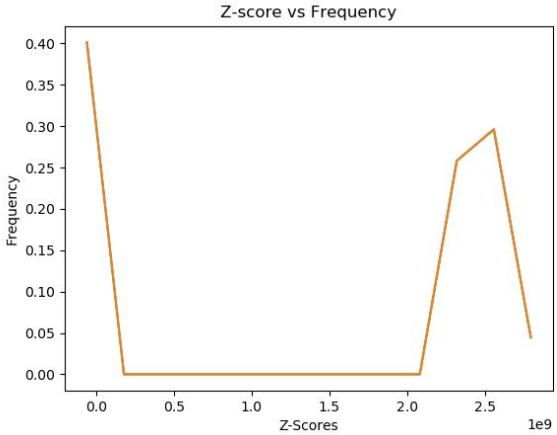
Minimum spanning tree of sample size 35000 (Cluster based-MST). Generated using Cytoscape. [6]



Zoomed in version of different sets







Runtime Analysis

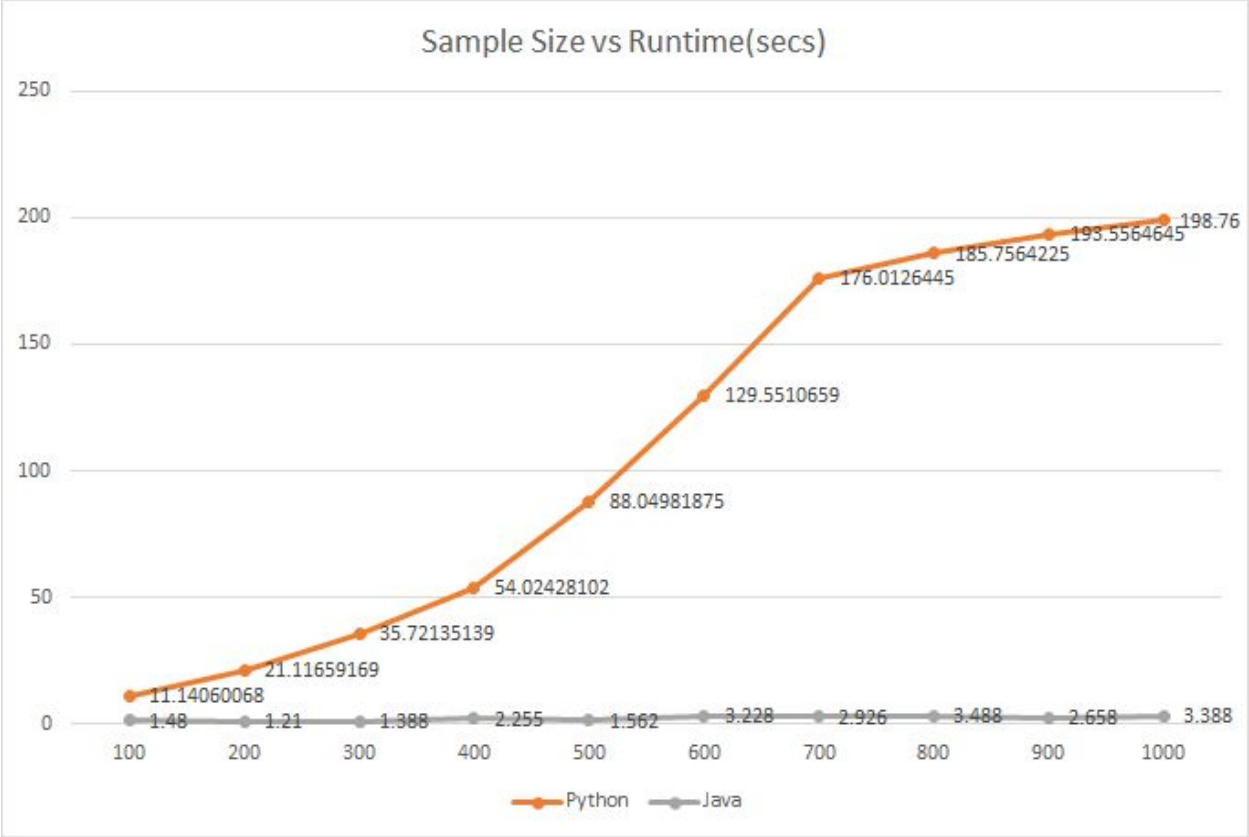
For a Sample Size of 1000, sMIN value of 3 and a sMAX value of 1000, the Python code took 198.76 seconds to calculate the E-Scores, but the Java Code consumed just 3.388 seconds for completing all sections to build the similarity network.

The efficiency can be calculated as follows

$$\frac{198.76-3.388}{198.76} * 100 = 98.29\ \%$$

Thus it can be seen that our implementation in Java is 98.29% faster than the original Python implementation.

The below graph shows the runtime comparisons for Java and Python and the execution time is measured in seconds against the sample values.



4. Discussions and Future Work

The main takeaway from this project is how we can find similarity in protein structures, molecules and even DNA with this particular Similarity Ensemble Approach. This approach is particularly useful with Large Datasets where we need clustering. This approach is really helpful in the booming Data Science fields where we require Clustering and similarity matching amongst huge datasets for the Analytics and Machine Learning.
We were also gratified with the results obtained and the process involved.

The previous implementation of the same project not assembled into proper packages, which led us to many pitfalls in the initial stage. But once we understood the approach, we were able to come up with our own methods of implementation.
Since the implementation was in Java, where there is less support for statistical packages and graphical interface unlike Python, writing entire Packages and libraries of Python in Java also proved to be an equally challenging task.

This work can be extended towards the implementation of the same by parallelization. Also, we need to consider the different domains the SEA could be utilized in. They involve Machine Learning, Bioinformatics, Psychology etc. Machine Learning or more preferably Deep Learning utilizes requires huge chunks of BigData in Gigabytes or even Terabytes. For instance, where

Supervised learning is required, there may arise the necessity of clustering to be formed. This will allow the data to be pre-processed and divided into proper classes.

5. Conclusion

This project demonstrates how we can improve the efficiency of finding the Similarity Sets with a simple Java Implementation. Similarity Ensemble Approach involves in finding chemical similarity sets amongst proteins and for target prediction in drug-related studies.. Though the partial implementation existed in Python, we were able to successfully convert the entire approach using Java, thereby decreasing the runtime. We were able to develop the Java Classes for the Python Libraries effectively. This involves the libraries such as Stats, Gumbel,Histogram and the Non-Linear Fitter. We were able to generate the Raw Scores and compare them with Expected E Values, and then able to use a Non Linear fitter to determine the exact polynomial fit. Later with the generated values of Raw Scores, we pass this to the Minimum Spanning Tree Class which follows the Kruskal's Algorithm to find the Similar Clusters. With the above approach we can conclude that SEA is an effective way to solve the clustering problem that exists in the world of Big Data, Data Science and Analytics.

The Similarity Ensemble Approach can be further parallelized to speedup sections 2.1, 2.2, and 2.3. Alterations to parallelize specific steps may provide good speedup depending on the machine environment.

Teamwork was one of the most pivotal aspects of the project. We were successful in dividing the tasks equally and were successful in implementing the same. As a Team we were able to learn how the Code is structured, the Test Cases are implemented and also how we need to maintain a Version Control for the Same. These are the team values that we intend to carry forward in the Future projects.

6. References

[1] Johnson, Mark A., and Gerald M. Maggiora. “Concepts and Applications of Molecular Similarity.” *Wiley.com*, 24 Sept. 1990.
[2] Keiser, M. J., & Hert, J. (2009). Off-target networks derived from ligand set similarity. In *Chemogenomics* (pp. 195-205). Humana Press, Totowa, NJ.
[3]https://commons.apache.org/proper/commons-math/javadocs/api-3.6/org/apache/commons/math3/fitting/PolynomialCurveFitter.html
[4]http://commons.apache.org/proper/commons-math/javadocs/api-3.6/org/apache/commons/math3/fitting/GaussianCurveFitter.html
[5] http://www.mathwave.com/articles/extreme-value-distributions.html#evd_gumbel
[6] https://cytoscape.org/
[7] https://www.geeksforgeeks.org/scipy-stats-gumbel_r-python/
[8] https://www.graphpad.com/guides/prism/7/curve-fitting/reg_how_to_gaussian.htm?toc=0&printWindow
[9] https://github.com/jakevdp/mst_clustering
[10] https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/