Optimization for Data Science Homework

Mojtaba Amini , Kamile Dementaviciute, Saeed Soufeh

May 2022

1 Introduction

Today, we are dealing with huge amounts of data in practical applications. Analyzing this type of data requires certain considerations when trying to solve any specific problem. Considering the many breakthroughs in the technology field, we still don't have enough computing power, so researchers introduced various new algorithms to increase the computation speed. One way of doing that is optimizing the update rule of gradient based methods in an specific ML problems to optimize the problem regarding to the boundaries.

In this study, semi-supervised learning classification problem will be considered and solved using three different methods: Gradient Decent, Randomized Block Coordinate Gradient Decent, and Cyclic Block Coordinate Gradient Decent. We will compare the efficiency of all three methods when considering different sample sizes, to see which works best in which specific case. All the implementations have been done in Python.

2 Generate the Data

First of all, all three methods were implemented on randomly generated labelled data using make blobs from scikit-learn library. Since this is a semi-supervised learning problem, 98% of labels were randomly picked and removed, while the remaining 2% were used in the training. In the plot below, you can see the distribution of labelled (black and red) as well as the unlabelled (blue) data points.

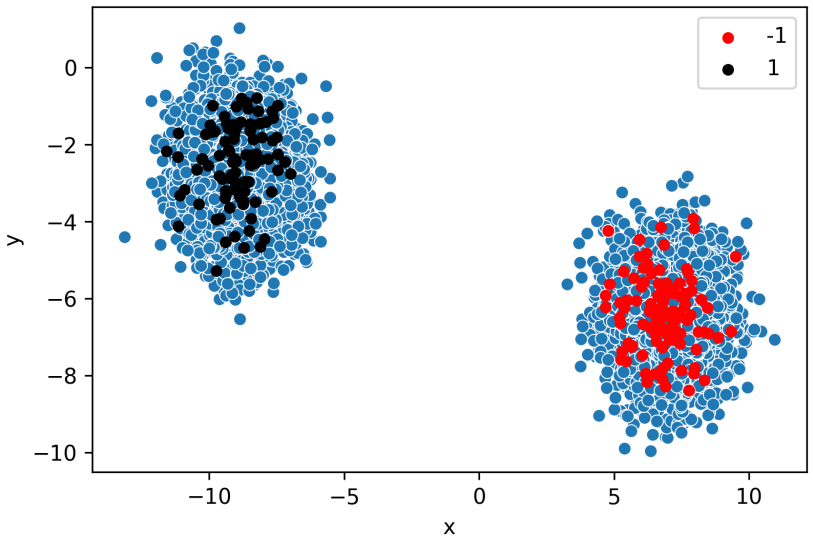

Figure 1: Randomly generated data: 2% (black & red) labelled, 98% (blue) do not have labels.

Related code:

```python
# generate 2d classification dataset
np.random.seed(15)
X, y = make_blobs(n_samples=300, centers=2, n_features=2)
x_Unlabeled ,x_labeled_bar,y1,y_label = train_test_split(X,y,test_size=0.02)

print(f"Original size of X= {X.shape[0]}\tUnlabeled size = {x_Unlabeled.shape[0]}\tlabeled size = {x_labeled_bar.shape[0]}")
print(f"Original size of y= {y.shape[0]}\tUnlabeled size = {y1.shape[0]}\tlabeled size = {y_label.shape[0]}")

y_label=y_label*2-1
df = DataFrame(dict(x=x_labeled_bar[:,0], y=x_labeled_bar[:,1], label=y_label))
colors = {-1:'red', 1:'black'}
fig, ax = pyplot.subplots()
grouped = df.groupby('label')
sns.scatterplot(ax=ax,x=X[:,0],y=X[:,1])
for key, group in grouped:
    group.plot(ax=ax, kind='scatter', x='x', y='y', label=key, color=colors[key])
pyplot.show()
```

Figure 2: Randomly generating and displaying the data in Python

3 Methodologies

Explanations of the problem and the different gradient decent methods used to solve it are given below.

3.1 Semi-Supervised problem

The cost function of a semi-supervised classification problem in this homework is written in the equation below (1). Additionally, the cost function can be easily derived as equation (2) with respect to any parameter j:

$$\min_{y \in R^u} \sum_{i=1}^{l} \sum_{j=1}^{u} \omega_{ij} (y^j - \bar{y}^i)^2 + \frac{1}{2} \sum_{i=1}^{u} \sum_{j=1}^{u} \bar{\omega}_{ij} (y^j - y^i)^2 \tag{1}$$

Gradient with respect to y^j:

$$\nabla_{y^j} f(y) = 2 \sum_{i=1}^{l} \omega_{ij} (y^j - \bar{y}^i) + 2 \sum_{i=1}^{u} \bar{\omega}_{ij} (y^j - y^i) \tag{2}$$

u : unlabeled data, l : labeled data, y : predicted label (parameters), $\bar{y}$: determined label, ω_{ij} : similarity measure matrix between labeled and unlabeled data, $\bar{\omega}_{ij}$: similarity measure matrix between unlabelled data.

3.2 Determining and Defining the Similarity Measure Function

The next step it to define the similarity measure ω_{ij}, $\bar{\omega}_{ij}$ is the similarity measure between two classes of the unlabelled data. There are different ways to define the similarity measure but it is necessary to consider their behavior before picking one of them. A kind of Gaussian distribution as equation (3) has been used as similarity measure function when solving similar problems. The results of this function are useful in this case as well, since it gives higher values for points that are closer together and smaller values for two points further apart. The results of the calculated similarity measure function also showed that for points of the same class, the distance between them is much smaller, so the weight regarding these two point in the similarity measure matrix is also higher. The similarity measure matrix has been defined as equation (4).

Therefore the similarity matrix is defined as below:

$$\omega_{ij} = e^{-(x_{1i}-x_{1j})^2+(x_{2i}-x_{2j})^2} \tag{3}$$

x_1 refers to the first dimension and x_2 is the second dimension in 2D, i, j are two indexes for labeled and unlabeled data.

$$\omega_{ij} = \begin{bmatrix} e^{-(x_{11}-x_{11})^2+(x_{21}-x_{21})^2} & e^{-(x_{11}-x_{12})^2+(x_{21}-x_{22})^2} & \dots \\ e^{-(x_{12}-x_{11})^2+(x_{22}-x_{21})^2} & \ddots & \dots \\ \vdots & \dots & \dots \end{bmatrix} \tag{4}$$

4 Implementation

The gradient decent algorithms are first-order iterative algorithms for finding minima of a differentiable function. The idea behind the gradient decent methods is picking the direction with the help of the gradient and taking a step in opposite direction of it. The difference in the methods explored further is the way of picking the direction and when as well as by which procedure the parameters are updated during the iterations. When the number of variables becomes large, it will be computationally expensive to calculate full gradients, which means classic gradient descent methods might not be efficient. Therefore, for finding optimal solution, the variable will be partitioned into a specific number of blocks, only one block will be considered for optimization.

Since the goal of the homework was comparing the three given algorithms, other optimization methods regrading to stopping condition and step size have been skipped in this study. Furthermore, the data was well separated, early stopping with a small fixed number of iterations was used. The reasoning behind this is saving computation time and the fact that the trends of the algorithms can be clearly seen given the chosen number of iterations. Additionally, it is more beneficial to compare the algorithms using the same number of iterations, as it clearly represents the speed of the algorithm given the same task. In addition, a fixed step size was used, the weights matrix was constant and it has been calculated before getting into iteration calculation.

4.1 Gradient descent (GD)

In the gradient decent, the parameter vector has been initialized with a random small value close to the zero and also the weight matrix has been calculated. The target value will also be updated in the for loop until reaching the maximum number of iterations.

```
time_tracking_GD = []

y_unlabel = initialize_parameters(x_Unlabeled)
wij,wij_bar = similarity(x_Unlabeled,x_labeled_bar,"Gaussian")
cost_func_GD = []

t_start = process_time()

for i in range(num_iterations):
  derivative = dev_function(wij,wij_bar,y_unlabel,y_label,method = "GD")
  y_unlabel = y_unlabel-alpha*derivative
  cost_func_GD.append(cost_function(wij,wij_bar,y_unlabel,y_label))
  time_tracking_GD.append(process_time() - t_start)

t_stop = process_time()
```

Figure 3: The implemented gradient descent procedure

4.2 Randomized Block Coordinate Descent (RBCGD)

For randomized BCGD the initializing and the calculation of the weight matrix is the same as for the classic Gradient Decent. The second for loop has been added to have the same number iteration like GD to have a better comparison among methods on calculation time. In addition to that, every time a direction will be picked with normal distribution for updating. In addition to that in the calculation part of derivative, only the considered block will be updated. The implementation of the RBCGD is presented in the figure 4.

```
time_tracking_RBCGD = []

y_unlabel=initialize_parameters(x_Unlabeled)
wij,wij_bar=similarity(x_Unlabeled,x_labeled_bar,"Gaussian")
cost_func_RBCGD = []

Block_size = 1
Num_Block = int(y_unlabel.shape[0]/Block_size)

t_start = process_time()
for i in range(num_iterations):
  for n in range(1,Num_Block+1):

    Block_index = np.random.randint(1,Num_Block+1)
    derivative = dev_function(wij,wij_bar,y_unlabel,y_label,method = "BCGD",index = Block_index-1,size = Block_size)
    y_unlabel[Block_index-1] = y_unlabel[Block_index-1] -alpha*derivative

  cost_func_RBCGD.append(cost_function(wij,wij_bar,y_unlabel,y_label))
  time_tracking_RBCGD.append(process_time() - t_start)


t_stop = process_time()
```

Figure 4: The implemented Randomized BCGD Procedure

4.3 Cyclic Block Coordinate Descent (CBCGD)

The only difference of cyclic BCGD is that here, each blocks will be used in a cyclic order for updating the parameters. The implementation of the cyclic BCGD is presented in the Figure 5.

```
time_tracking_CBCGD = []

y_unlabel=initialize_parameters(x_Unlabeled)
wij,wij_bar=similarity(x_Unlabeled,x_labeled_bar,"Gaussian")
cost_func_CBCGD = []

Block_size=1
Num_Block=int(y_unlabel.shape[0]/Block_size)


t_start = process_time()

for i in range(num_iterations):
  for Block_index in range(1,Num_Block+1):

    derivative = dev_function(wij,wij_bar,y_unlabel,y_label,method = "BCGD",index = Block_index-1,size = Block_size)
    y_unlabel[Block_index-1] = y_unlabel[Block_index-1] -alpha*derivative

  cost_func_CBCGD.append(cost_function(wij,wij_bar,y_unlabel,y_label))
  time_tracking_CBCGD.append(process_time() - t_start)

t_stop = process_time()
```

Figure 5: The implemented Cyclic BCGD Procedure

5 Results

5.1 Comparing the Results of Different Algorithms

We use different number of samples (n : 300, 1000, 3000, 5000, 7000, 10000) to see which algorithm has the best performance with differing sample sizes.

As you can see in the figures below, The gradient descent has a better performance when the sample size is smaller (for instance, when n is equal to 300). When the sample size increases, cyclic block coordinate gradient descent gives a better performance, as it converges faster.

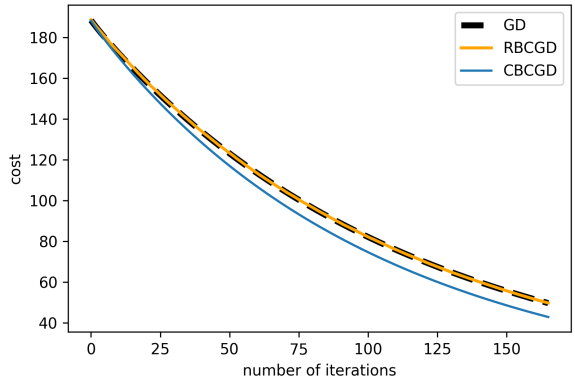

Figure 6: n = 300

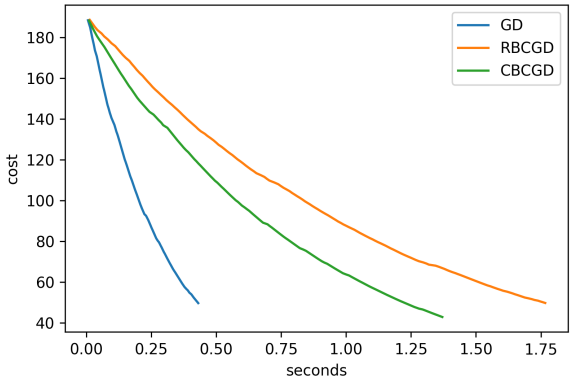

Figure 7: n = 300

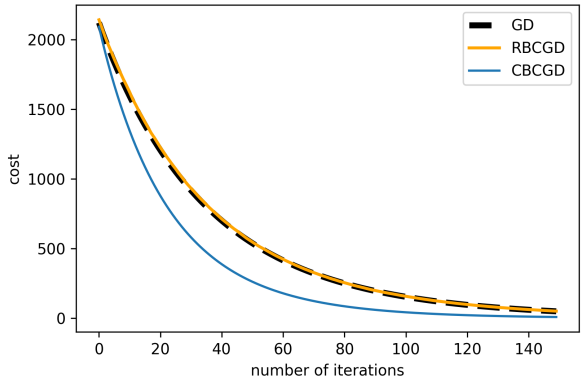

Figure 8: n = 1000

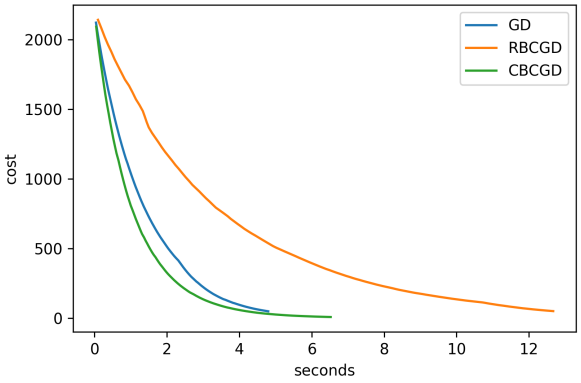

Figure 9: n = 1000

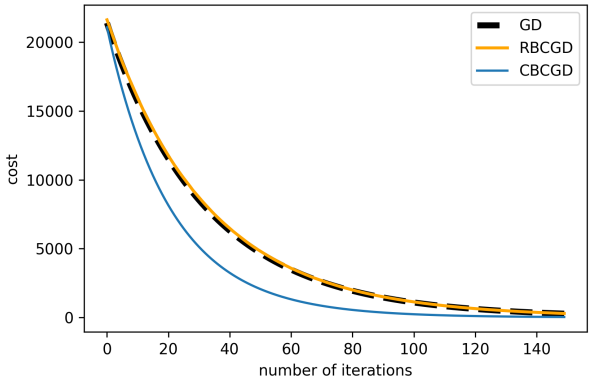

Figure 10: n = 3000

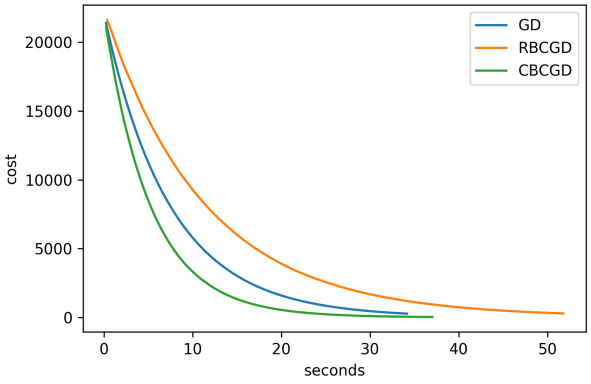

Figure 11: n = 3000

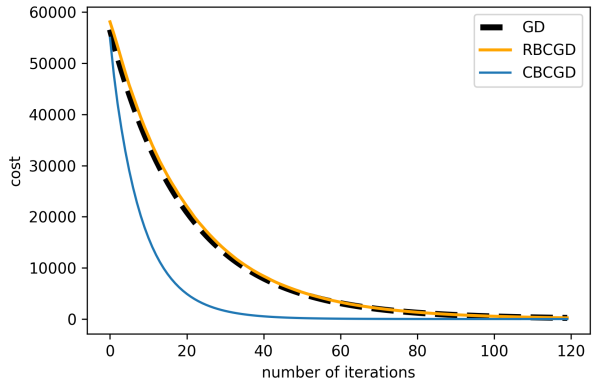

Figure 12: n = 5000

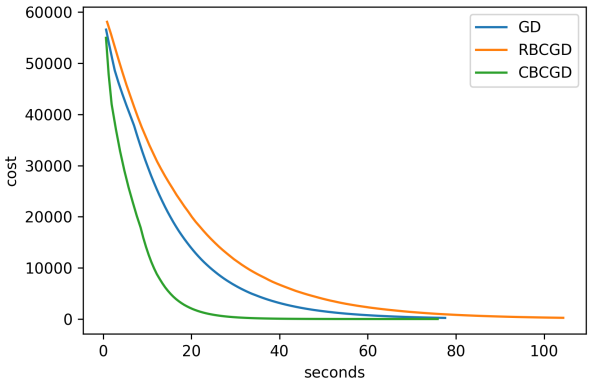

Figure 13: n = 5000

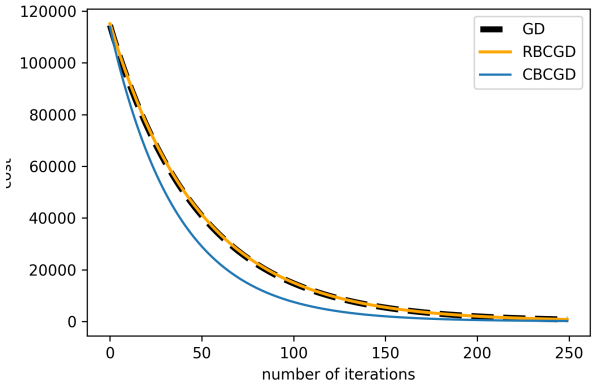

Figure 14: n = 7000

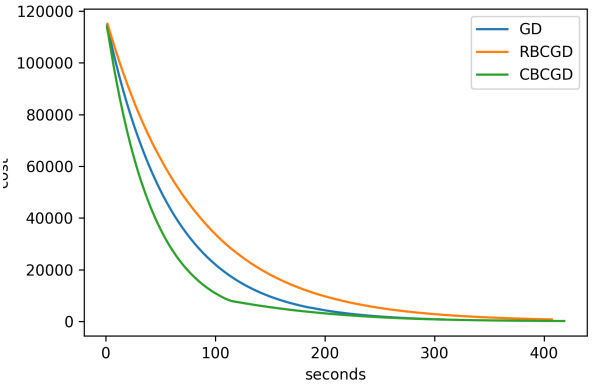

Figure 15: n = 7000

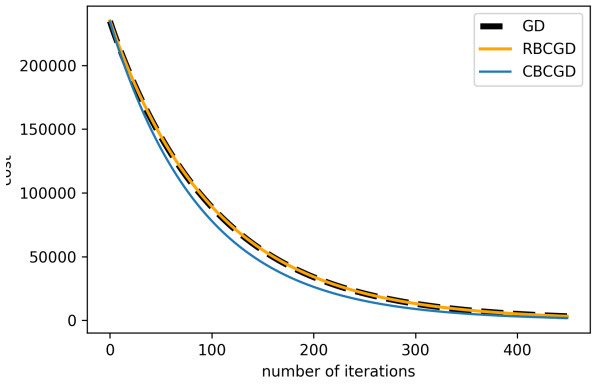

Figure 16: n = 10000

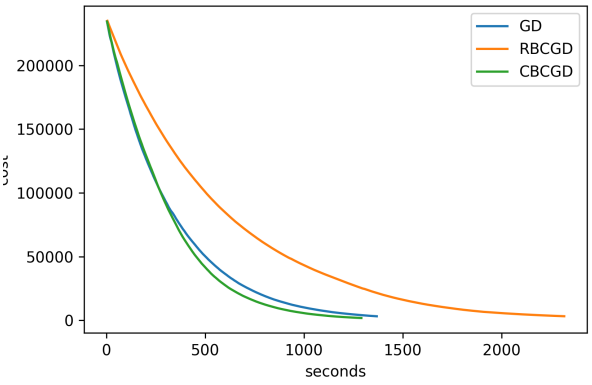

Figure 17: n = 10000

6 Addition tests on an outsourced data set

Additionally to randomly generated data, we wanted to see how the three methods perform on an outsourced dataset. For this we used Spambase dataset from UCI Machine Learning Repository. The

dataset includes instances of two class emails: spam and not spam. The number of features is 57 and the number of instances: 4601.

Having run all three methods for the same number of iterations (300), we see similar results to the previously demonstrated ones with sample size 10,000. Although the current sample size is smaller, the results are also influenced by the larger number of features. Figure 19 demonstrates the CPU time, required to run each method for 300 iterations. Gradient Decent and Cyclic Block Coordinate Gradient Decent show similar CPU time requirements, with CBCGD taking slightly less time, while Randomized Block Coordinate Gradient Decent clearly is the least efficient method in this case.

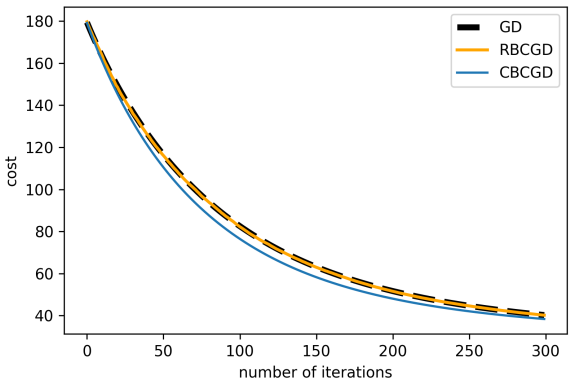

Figure 18: Testing with Spambase dataset

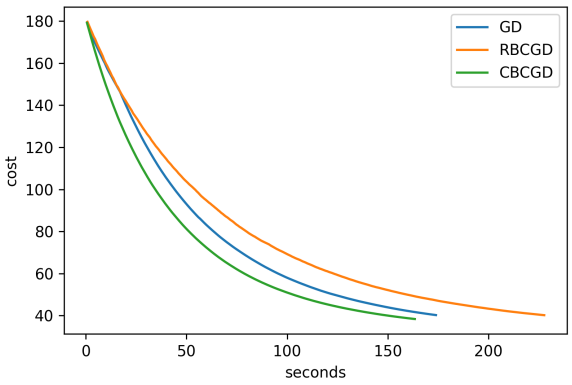

Figure 19: Testing with Spambase dataset

7 Conclusion

Having seen the implementation and convergence results of the three methods on datasets with varying sample sizes and a different number of features, we can conclude that both of them influence which method will perform best. As it can be seen in the Figure 20, with the increasing sample size and the number of features, the Cyclic Block Coordinate Gradient Decent method becomes more efficient than than classic Gradient Decent. It is important to note that a large amount of zeros in the similarity matrix 21 contribute to a significantly worse result from the RBCGD. Similarly, after running the methods with the outsourced dataset which includes a much larger number of features, the same trends can be observed.

For future investigation, it would be beneficial to carry out in depth exploration of different similarity measures as well as using exact line search to get a better performance from BCGD methods.

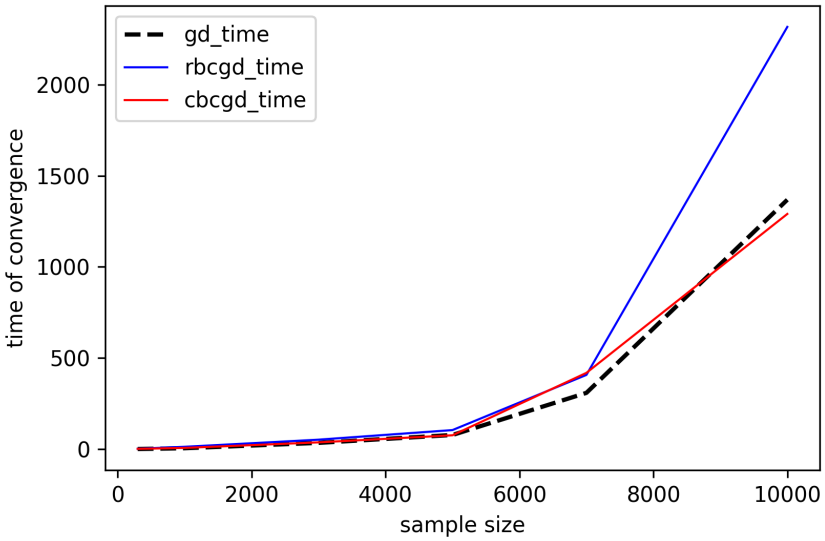

Figure 20: Comparing the result of methods in different data size.

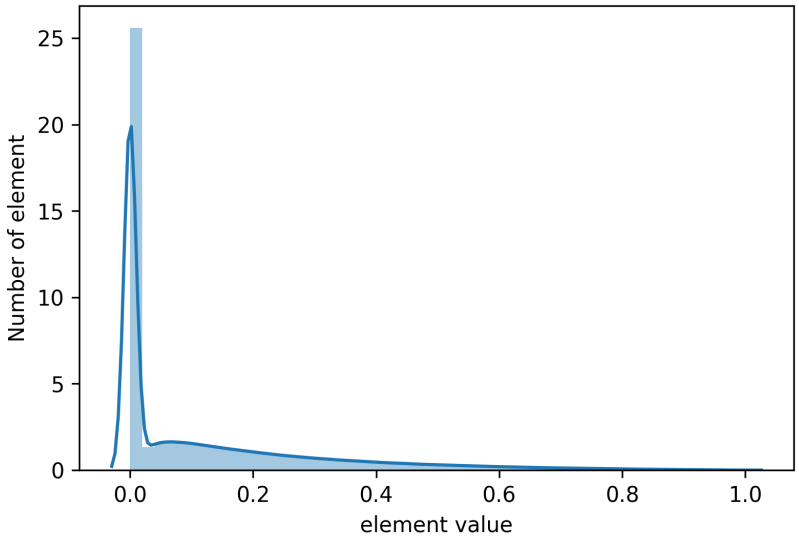

Figure 21: Similarity matrix values