DSC_Assignment-2

Aditya gupta(2022031) Sahil Gupta(2022430)

November 25, 2024

1 Question-1

In this question , we need to test different hashing functions and check the lengths of the maximum and minimum , also we have to estimate the number of unique words in the list of ids of the words in word.txt .

1.1 Part - A

In this part of the question , we have implemented the said hash table data structure class which is the normal linked list type of data structure in a hash table . The constructor of the class is as follows :

```
def __init__(self , hash_function_used ,  size: int=16 , prime: int=524287):
```

Thus the initializations are made as said in the question . Also the we have created a hash function which maps element x to ((ax) mod p) mod m) . Thus this has been used in the part (c) of this question . The code for the same is in the coding file q1.ipynb .

1.2 Part - B

Now, in this part of the question, we need to create a hash function for the strings to map them to buckets. We use the `md5` library and, using the `hexdigest()` method, we get the hash values for each word. We then only take the last 4 digits of this hex-digested mapping. Thus, in total, we have:

$$16^4 = 65536$$

possible buckets. We are storing these values in hash_values_stored . We are also storing the encoded values in a new file named "word_new.txt" . This is the things to be done in this part of the question .

1.3 Part - C

In this part of the question , we need to experiment for the maximum chain length for two hash functions . The first one is universal hash function which was created in the part a of this assignment and the other is random hash function . We need to create 500000 buckets (m = 500000) . Now for each number from 1 to m , we need to insert it into the the hash function and check for the maximum and the minimum size of each bucket . We need to do this experiment for 5 number of iterations . Now , we do this for 5 times and the graphs for the same is as follows :

First For the case of Universal Hashing :

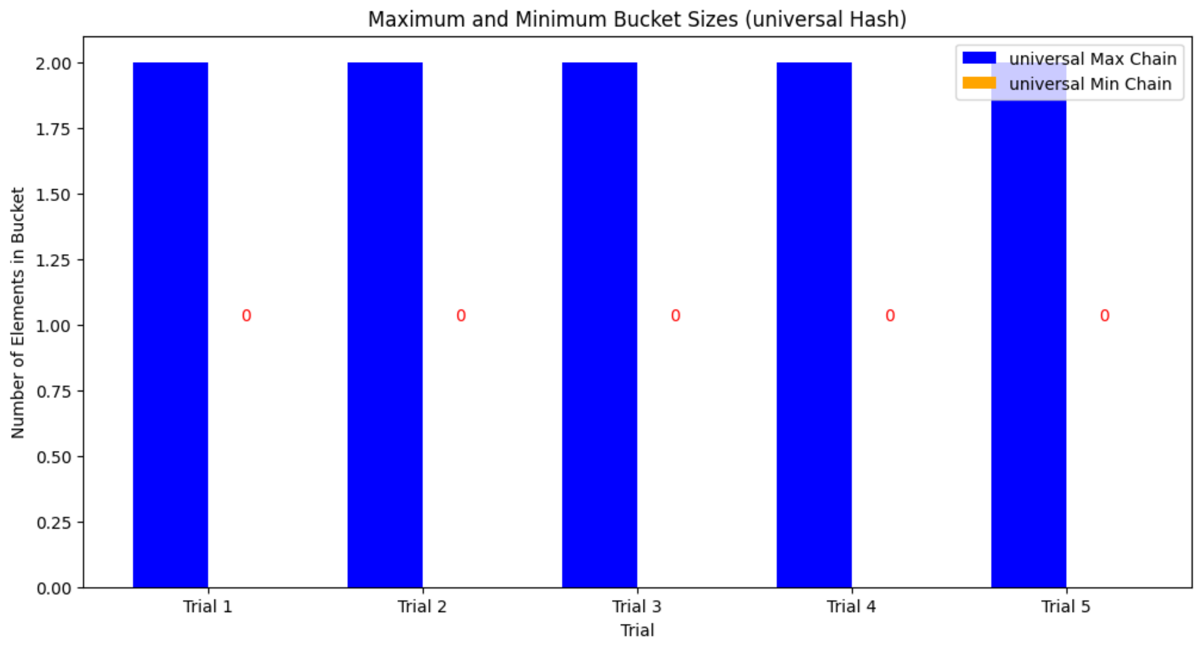
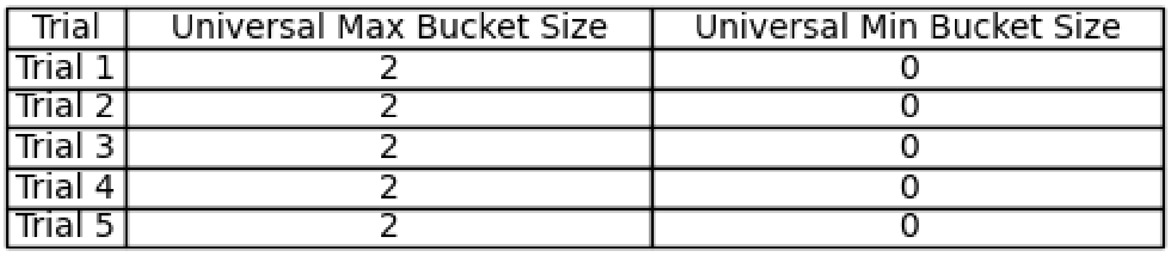


Figure 1: Max_Min_Universal

We can see that for Universal Hashing , the max is coming to be around 2 elements and the minimum elements to be 0 elements in a buckets , thus this shows that there is very less difference betweeen min and max elements in a bucket in the Universal Hashing case . The table for Universal Hashing is as follows :



Trial	Universal Max Bucket Size	Universal Min Bucket Size
Trial 1	2	0
Trial 2	2	0
Trial 3	2	0
Trial 4	2	0
Trial 5	2	0

Figure 2: Universal_Hashing_Table

Secondly , for the case of Random Hashing , we have the following :

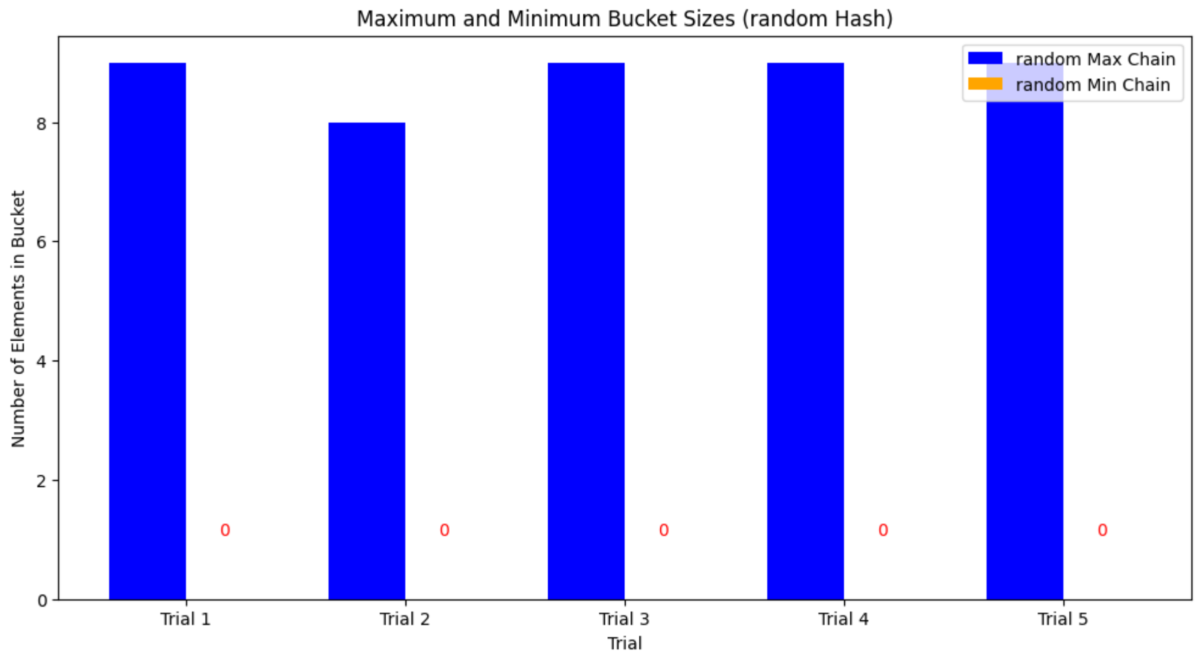
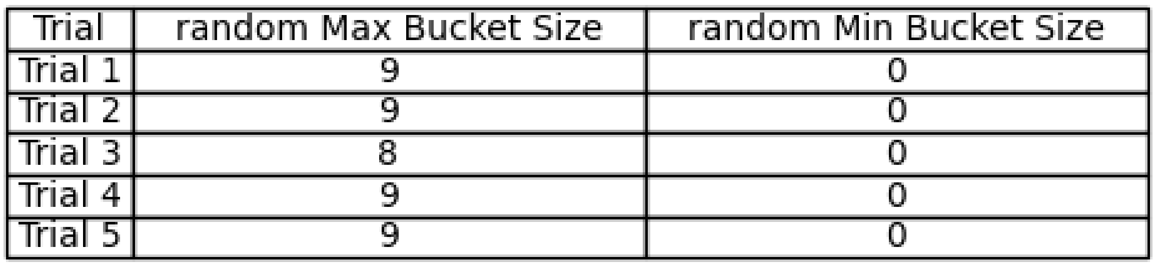


Figure 3: Max_Min_Random

We can see that for Random Hashing , the max is coming to be around 8 to 9 elements and the minimum elements to be around 0 elements in a bucket, thus this shows that there is a lot more difference between the min and max bucket in the case of Random Hashing Function showing that this is not good for hashing purposes as the length of a chain can get a lot bigger than the Universal Hashing function . The table for random Hashing Function is as follows :



Trial	random Max Bucket Size	random Min Bucket Size
Trial 1	9	0
Trial 2	9	0
Trial 3	8	0
Trial 4	9	0
Trial 5	9	0

Figure 4: Random_Hashing_Table

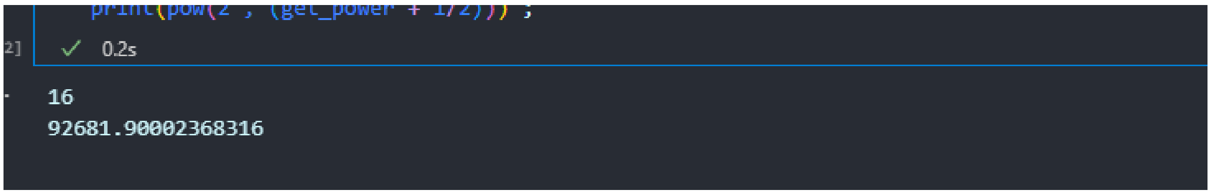
Thus , Random Hashing had more difference between the maximum and the minimum chain . This ends this part .

1.4 Part - D

In this part of the question , we need to use the id of every word in the file "word.txt" , we formed in part (b) of this question . Now , we have to use m=500000 .Now , we use the universal hasing function from problem (a) to get the hash values of these ids . Now we get the hashed values , we now get the maximum number of trailing zeros in the binary notation of these hashed values . Storing them as Z , we return

$$16^{(}z + 1/2)$$

The result in this case varies due to the many random factors like the value of a being random from 0 to (p-1) . One such example is as follows :



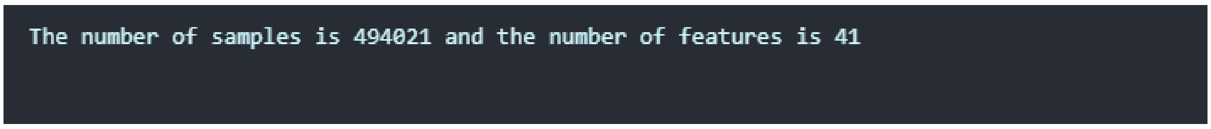
```
print(pow(2 , (get_power + 1/2))) ;
✓ 0.2s

16
92681.90002368316
```

Figure 5: Estimated_Values

Thus this is the result for part d of this question with the required things done .

2 Question-2

Now in this part of the assignment we need to use random projection on the KDD Cup dataset . First we need to compute the number of samples as n and number of features as d . The values of it are as follows :



```
The number of samples is 494021 and the number of features is 41
```

Figure 6: Data

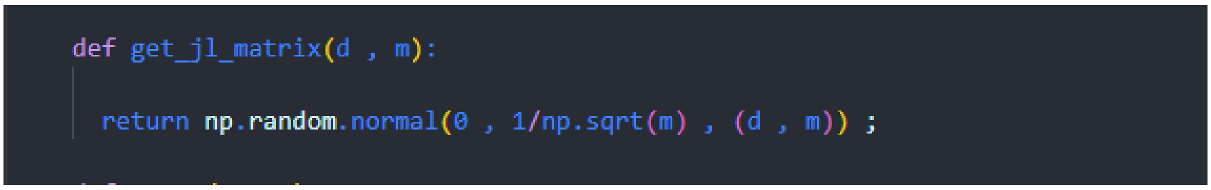
Thus this is what we get the dimensions of the data ,

```
data belongs to R^ (n*d) , where n = 494021 and d = 41 .
```

Thus now we can move to the next parts of the question .

2.1 Part - A

In this part of the question , we have to define a family of JL matrices of size d * 20 . We do this by the following code :



```
def get_jl_matrix(d , m):

    return np.random.normal(0 , 1/np.sqrt(m) , (d , m)) ;
```

Figure 7: family_JL_PART_A

Thus the entries from the JL matrix is from a gaussian RV with mean 0 and variance 1/m , where m is 20 as given in the question . Now we make the matrix E = DM , thus E has dimension of n*20 or 494021 * 20 . Thus , we get D which is the original matrix and E which is the newly formed matrix . Now , we find centroids with k = 15(clusters are 15) and using these centroids find lose on the dataset

D . Now , the dimensions of the centroids from E is 20 , while the datapoints in D are of dimension 41 . Thus , we have use pseudoinverse of M and dot product it with centroids of E to get them back in the higher dimension. Thus we get the loss for 5 iterations as follows :

```
Experiment Results Table is as follows :
Experiment | Loss on D (original) | Loss on DM (reduced)
     1     |    3228505318238.99          |    162992148546190464.00
     2     |    3228505318238.99          |    224548169983508224.00
     3     |    3228505318238.99          |    254358074768816832.00
     4     |    3228505318238.99          |    160546098080768800.00
     5     |    3228505318238.99          |    278990341529104928.00
```

We can see that the loss due to E = DM is way higher than Loss due to D . This is mostly due to loss of data when reducing the features dimensions . Thus getting higher loss . This can also be visualized in the graphs as follows :

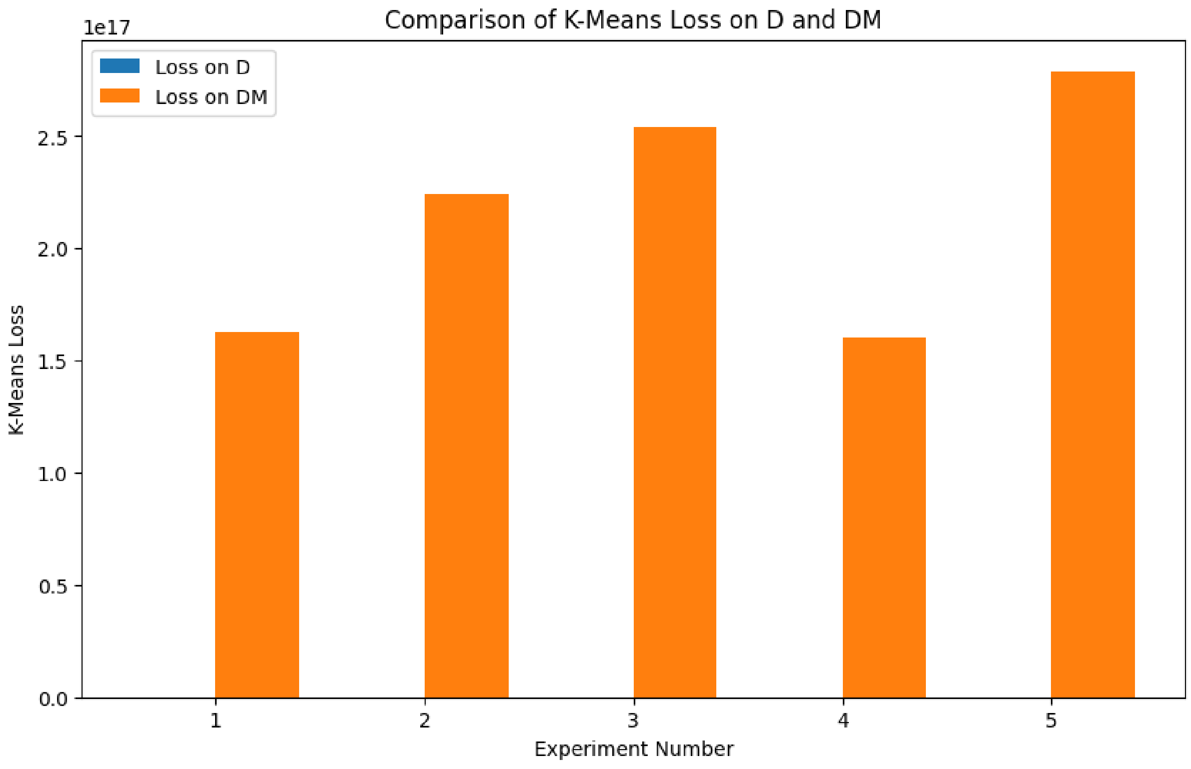
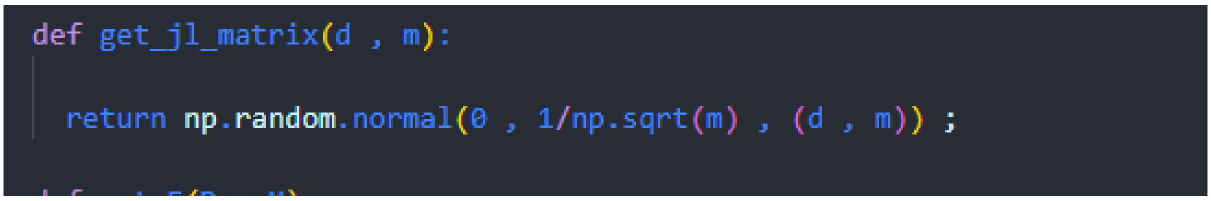


Figure 8: k_means_loss

The loss due to D is very small as compared to E=DM as we can see from the graph also . Thus we did this experiment 5 times with different values of M to obtain these results .

2.2 Part - B

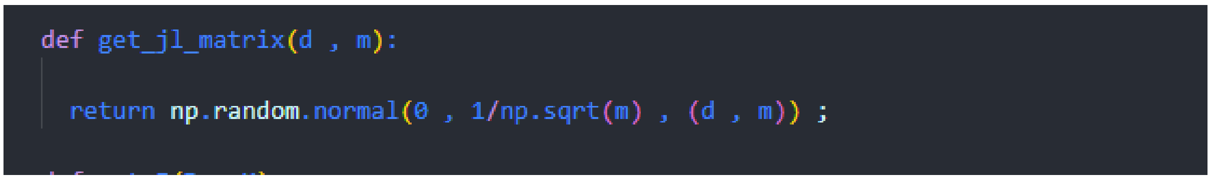
Now in this part of the question, we need to run linear regression on the dataset . For this we have the label feature as y which is of size 23 or it has 23 classes . Now we define a family of JL matrices if size 10d * n as follows :



```
def get_jl_matrix(d , m):
    return np.random.normal(0 , 1/np.sqrt(m) , (d , m)) ;
```

Figure 9: jl_matrix_normal

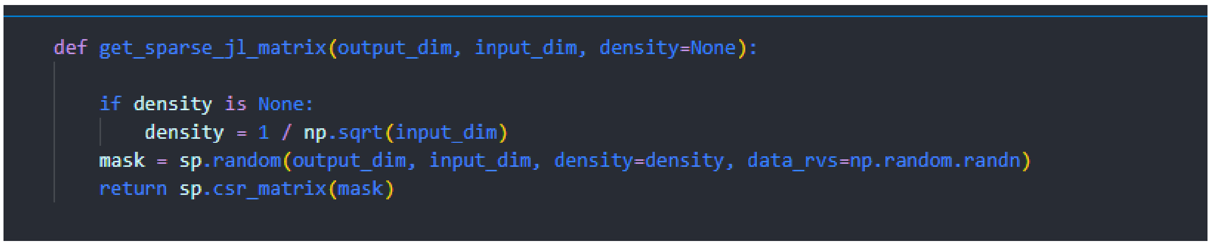
The JL family without using sparsity is defined as follows :



```python
def get_jl_matrix(d , m):
    return np.random.normal(0 , 1/np.sqrt(m) , (d , m)) ;
```

Figure 10: Without_Sparsity

In this d = 10*d and m = n in this case . This is for without implementing the sparsity . Now we also implemented sparsity for faster computation and as said in the question , we made the density 1/root(n) , which is the default sparsity to take . Thus this reduces size from n to root(n) of the dataset , reducing computation time . The code for it as follows :



```python
def get_sparse_jl_matrix(output_dim, input_dim, density=None):
    if density is None:
        density = 1 / np.sqrt(input_dim)
    mask = sp.random(output_dim, input_dim, density=density, data_rvs=np.random.randn)
    return sp.csr_matrix(mask)
```

Figure 11: With_Sparsity

Thus this is the jl_matrix family with sparsity . Thus we get the matrices E = MD and Z = MY , thus the dimensions of E is (10*d) * d and that of Z is (10*d) * 1. Thus ,we train linear regressors on Data E and labels Z and on Data D and labels Y . Thus after this , we use these regressors on the original Data D and the results are as follows :

First without sparsity in JL matrix :

```
    Experiment Results Table is as follows :
Experiment | Loss on E_Z | Loss on D_y
     1     |    0.94          |    0.82
     2     |    0.94          |    0.82
     3     |    0.90          |    0.82
     4     |    0.92          |    0.82
     5     |    0.94          |    0.82
```

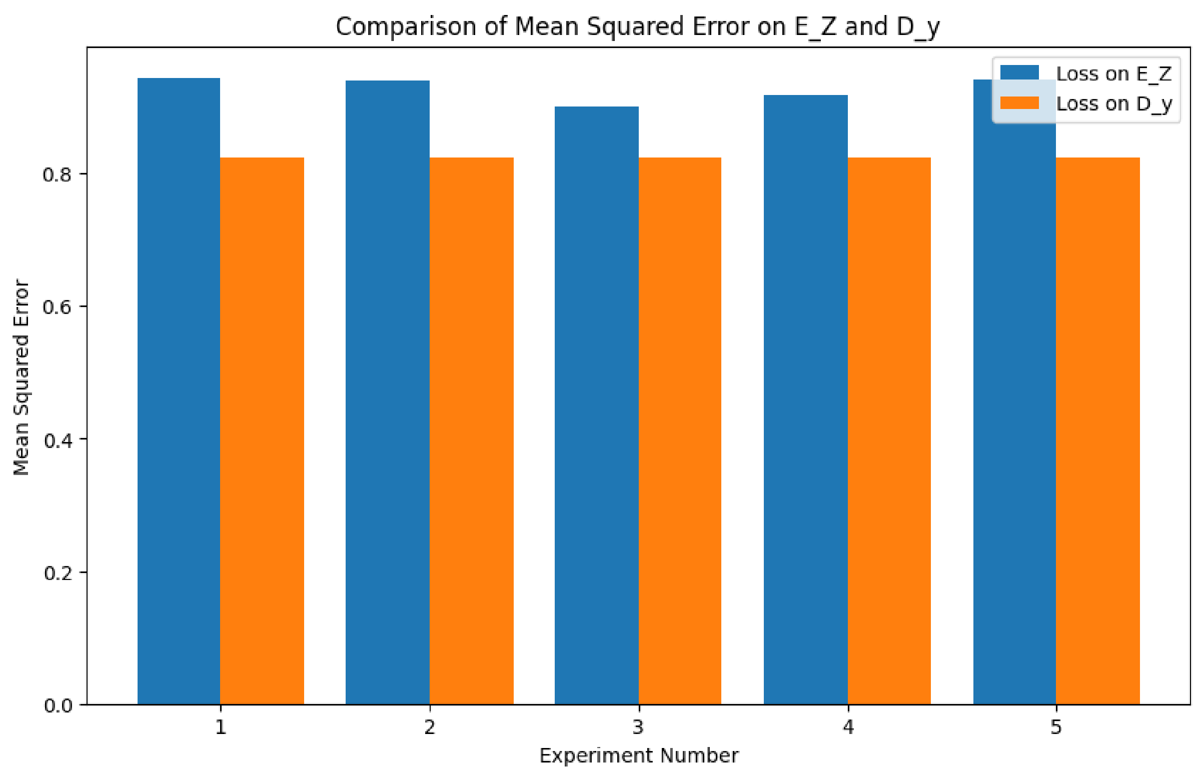


Figure 12: plot_without_sparse

Thus we can see that the regressor formed from (E,Z) gives higher loss than the one trained on (D,Y) . This is due to loss of data when we go from higher dimension(n) to lower dimension(10*d) .

Now for the case of sparse JL matrix family , we have the following tables and plots :

```
    Experiment Results Table is as follows :
Experiment | Loss on E_Z | Loss on D_y
     1     |    1.86          |    0.82
     2     |    12.12          |    0.82
     3     |    7.67          |    0.82
     4     |    2.37          |    0.82
     5     |    2.92          |    0.82
```

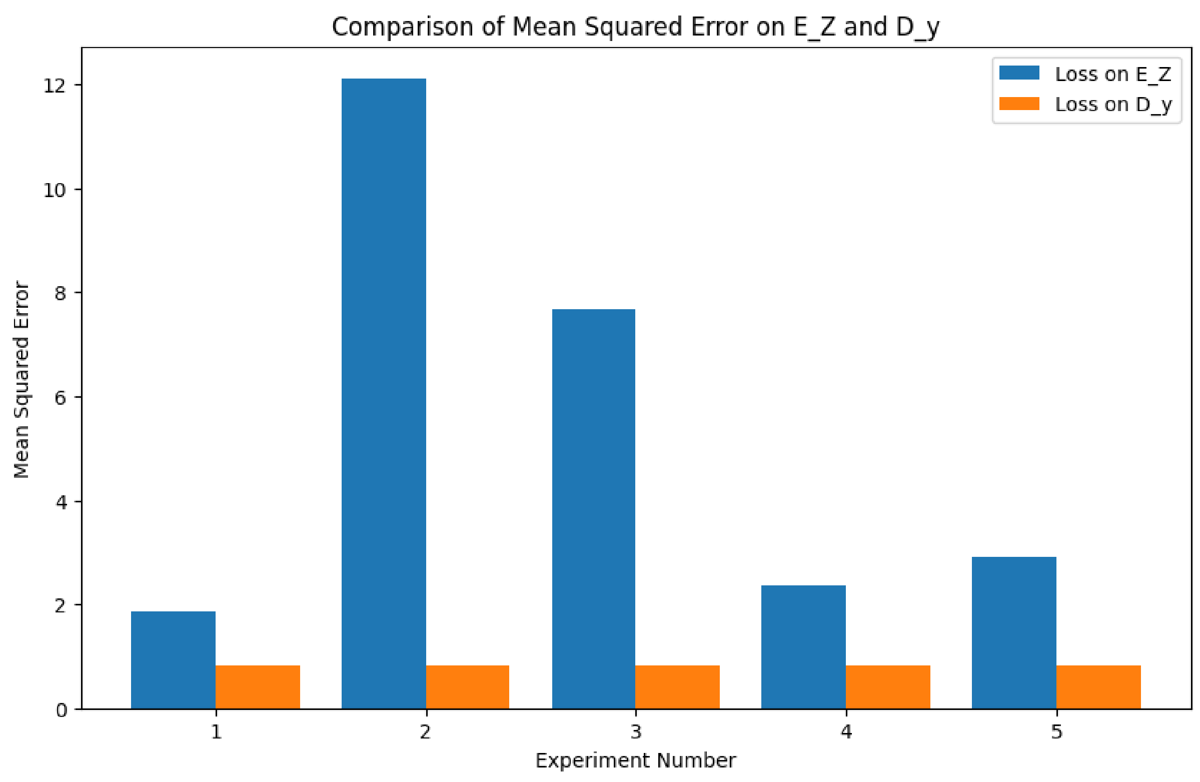


Figure 13: Plot_Sparse

Thus in this also the loss even increase more than before in the case of (E,Y) as even few data points are taken into consideration in the sparse JL matrix family . This was the whole analysis for the assignment .