Sreyashi Saha Matriculation No: 1747271

Distributed Machine Learning (Supervised)

In this exercise sheet we are going to implement a supervised machine learning algorithm in a distributed setting using MPI (mpi4py). We will pick a simple Linear Regression model and train it using a parallel stochastic gradient algorithm (PSGD).

Dataset

I have used the following dataset for this task.

- DynamicFeaturesofVirusShareExecutablesDataSethttps://archive.ics.uci.edu/ml/datasets/ Dynamic+Features+of+VirusShare+Executables

STEP1: READ THE TEXT FILES AND CONVERT THEM INTO A CSV FILE

```
rows = []
for file in tqdm(os.listdir('dataset')):
    for line in open(f"dataset/{file}", 'r'):
        features = line.split()
        target = features[0]
        row = {feature.split(':')[0]: feature.split(':')[1] for feature in features[1:]}
        row['target'] = target
        rows.append(row)

df = pd.DataFrame(rows)
new_columns = {int(col) for col in df.columns if col != 'target'}
new_columns = list(map(str, new_columns)) + ['target']
df = df.reindex(new_columns, axis=1)
df = df.replace(np.nan, 0)
df.to_csv('/Users/sreyashisaha/Desktop/Semester1/DDA/practical/Week5/virus.csv',index=False)
```

Here I read all the 45 text files and store the data into a data frame. Since this data had a lot of NAN values, I therefore, replaced the NAN values with ZEROS. I then exported this newly filled Dataframe into a CSV file for further implementation.

STEP2: SPLIT THE DATASET (RATIO - 70:30)

```
def train_test_split(data):
    np.random.seed(0)
    split_ratio = np.random.rand(len(data))<0.7
    train_data = data[split_ratio]
    test_data = data[~split_ratio]
    return train_data, test_data
```

In this function the variable split ratio stores all those values for which the random number assigned to each row is less than 0.7. It equally divide the data set into a 70:30 ratio. The variable basically stores values in boolean, i.e. it stores true for values less than 0.7 and False for those greater than 0.7. Then we extract all the rows which has has value less than 0.7 and store it in the train dataset and the rest of the values in the test dataset.

```
sreyashisaha@Sreyashis-MacBook-Air Week5 % python3 try3.py
SPLIT RATIO [ True False  True ...  True  True False]

SPLIT RATIO [ True False  True ...  True  True False]

X Shape:  (107856, 266)
Y Shape:  (107856,)
X_train Shape:  (75444, 266)
Y_train Shape:  (75444,)
```

STEP2: NORMALISING DATA

The process of transforming data into a consistent format so that users may process and evaluate it is known as data normalization. Most businesses use data from a variety of places, including data warehouses, lakes, cloud storage, and databases. Data from many sources, on the other hand, can be problematic if it isn't uniform, resulting in problems down the road (e.g., when we use that data to produce dashboards and visualizations, etc.).

For a variety of reasons, data normalization is essential. First and foremost, it assists us in establishing clearly defined elements and attributes, resulting in a thorough catalog of our data. Whatever insights or problems we're attempting to address, a thorough grasp of our data is a necessary first step.

To get there, we'll need to convert the data into a standard format with logical and consistent definitions. These specifications will be used to create metadata, which are labels that describe what, how, why, who, when, and where our data is stored. Our data standardization procedure is built on this foundation.

Normalizing the way we label data will increase access to the most relevant and updated information from the standpoint of accuracy. This will make our statistics and reporting more straightforward.

I have used the following method to normalize the dataset.

```
# ---------- standardizing data----------
norm_matrix_traindata = np.linalg.norm(x_train)
normalised_array_train = x_train / norm_matrix_traindata

norm_matrix_testdata = np.linalg.
normalised_array_test = x_test / norm_matrix_testdata
```

STEP3: IMPLEMENTING SGD

The iterative approach of stochastic gradient descent (commonly abbreviated SGD) is used to optimize an objective function with sufficient smoothness criteria (e.g. differentiable or subdifferentiable). Because it replaces the actual gradient (derived from the complete data set) with an estimate, it can be considered a stochastic approximation of gradient descent optimization (calculated from a randomly selected subset of the data). This minimizes the extremely high computational cost, especially in high-dimensional optimization problems, allowing for faster iterations in exchange for a reduced convergence rate.

For eg: Say my model is y_hat = w1*x1 + w2*x2 + b, (w1,w2 = weights, b=bias)

My loss function is L = (y_hat - y_actual)^2. Because we want the difference between $\hat{y}$ and y to be small, we want to make an effort to minimise it. This is done through **stochastic gradient descent** optimisation. It is basically iteratively updating the values of w_1 and w_2 using the value of gradient, as in this equation:

$$w_{\text{new}} = w_{\text{current}} - \eta \frac{\partial L}{\partial w_{\text{current}}}$$

This algorithm tries to find the right weights by constantly updating them, bearing in mind that we are seeking values that minimise the loss function.

There are many ways to initialise weights (zeros, ones, uniform distribution, normal distribution, truncated normal distribution, etc.) and here I have initialised the weights and bias with zero.
I have divided our dataset into smaller groups of equal size. Each group is called a **batch** and consists of a specified number of examples, called **batch size**. If we multiply these two numbers, we get back the number of observations in our data.

Before we start adjusting the values of the weights and bias w_1, w_2 and b, let's first compute all the partial differentials. These are needed later when we do the weight update.

We complete 1 epoch when the model has iterated through all the batches once. In practice, we extend the epoch to more than 1.

One **epoch** is when our setup has seen **all** the observations in our dataset once. But one epoch is almost always never enough for the loss to converge. In practice, this number is manually tuned.

```
def MyCustomSGD(train_data,test_data,y_test,learning_rate,n_iter,k,divideby):

    # Initially we will keep our Weights and Intercept as 0 as per the Training Data
    w=np.zeros(shape=(1,train_data.shape[1]-1))
    b=0
    train_data = pd.DataFrame(train_data)
    cur_iter=1
    local_mse = []
    timeList = []
    while(cur_iter<=n_iter):
        # We will create a small training data set of size K
        temp=train_data.sample(k)
        # print("temp is",temp)
        # We keep our initial gradients as 0
        w_gradient=np.zeros(shape=(1,temp.shape[1]-1))
        b_gradient=0
        # temp.to_numpy()
        # We create our X and Y from the above temp dataset
        y=np.array(temp.iloc[:,-1])
        x=np.array(temp.iloc[:,0:265])
        # print(x)
        for i in range(k): # Calculating gradients for point in our K sized dataset
            prediction=np.dot(w,x[i])+b
            w_gradient=w_gradient+(-2)*x[i]*(y[i]-(prediction))
            b_gradient=b_gradient+(-2)*(y[i]-(prediction))
        #Updating the weights(W) and Bias(b) with the above calculated Gradients
        w=w-learning_rate*(w_gradient/k)

        b=b-learning_rate*(b_gradient/k)
        # print("Intercept is",b)

        # Incrementing the iteration value
        cur_iter=cur_iter+1

        #Dividing the learning rate by the specified value
        learning_rate=learning_rate/divideby
        y_pred_epoch = predict(test_data, w, b)
        rmse_ranks = RootMeanSquare_Result(y_test, y_pred_epoch)
        local_mse.append(rmse_ranks)
        # print(local_mse)
    # print("Coefficients",w)
    return w,b, local_mse #Returning the weights and Intercept
```

STEP4: CALCULATING LOSS FUNCTION

```
def predict(x_test,w,b):
    y_pred=[]
    x_test = np.delete(x_test, -1, axis=1)
    # print("New Shape",x_test.shape)
    for i in range(len(x_test)):
        y=np.asscalar(np.dot(w,x_test[i])+b)
        y_pred.append(y)
    return np.array(y_pred)
```

First I predict the y values using the test dataset. Later I calculate RMSE which is my loss function in this case.

```
def RootMeanSquare_Result(y_test_actual,y_test_pred):
    MSE = np.square(np.subtract(y_test_pred,y_test_actual)).mean()
    # print("MSE is",MSE)
    RMSE = math.sqrt(MSE)
    # print("Root Mean Square Error:\n")
    # print(RMSE)
    return RMSE
```

STEP5: WORK IN PARALLEL

In the master node I have first read my csv file and initialised the dependent and independent variable. In this was the last column from the dataset is my dependent or target variable and the remaining columns are my covariates or my independent variables. Since from the dataset it is not very clear what each of the covariates signify, I have therefore not dropped any of the columns from my training or testing dataset.

```
if rank == root:
    # Loading Data from CSV File
    file_data = pd.read_csv('virus.csv')
    X = file_data.iloc[:, 0:265].to_numpy()
    Y = file_data.iloc[:, -1].to_numpy()
    # Split Data into train and test
    x_train, x_test = train_test_split(X)
    y_train, y_test = train_test_split(Y)

    print("X Shape: ", X.shape)
    print("Y Shape: ", Y.shape)
    print("X_train Shape: ", x_train.shape)
    print("Y_train Shape: ", y_train.shape)
```

After initialising my dependent and independent variables I have split my data in the ratio of 70:30. I have also printed the shapes so as to get a clear understanding.

```
X Shape:  (107856, 265)
Y Shape:  (107856,)
X_train Shape:  (75444, 265)
Y_train Shape:  (75444,)
```

After this step I have normalised my data by calling numpy.linalg.norm(arr) to find the normal form of an array arr. Normalising basically means dividing the vector by its norm. It refers to rescaling by the minimum and range of vector, to make all the elements lie between 0 and 1, thus bringing all the values of the numeric columns in the dataset to a common scale.

```
    # ------------Adding the target Column in the training data---------
    train_data = pd.DataFrame(normalised_array_train)
    train_data['target'] = y_train
    x_test = np.array(normalised_array_test)
    y_test = np.array(y_test)
    # split array in the number of workers opened
    array_split = np.array_split(train_data, size)
else:
    x_test = None
    y_test = None
    split_array_train_data = None
```

Next I have split my entire training data according to the number of processes.

```
train_data = comm.scatter(array_split, root=root)
print(f'rank:{rank} got this size of training Data:{len(train_data)}')
x_test = comm.bcast(x_test, root=root)
y_test = comm.bcast(y_test, root=root)
```

I have then sent my training data to all the ranks and broadcasted the testing datas as well.

After broadcasting and scattering the datasets, I now need to gather all the weights and intercepts to plot the performance graph.

```
w, b, RMSE_gathered= MyCustomSGD(train_data, x_test, y_test,learning_rate=0.01, n_iter=200,k=50,divideby=1)
rmse_each_epoch_Gather = comm.gather(RMSE_gathered, root=root)
# print('msegather:', rmse_each_epoch_Gather,'\n')
w_reduced = comm.reduce(w, root=root)
b_reduced = comm.reduce(b, root=root)
# print('w_reduced:', w_reduced,'\n')
# print('b_reduced:', b_reduced,'\n')
```

These are the final summed up weights and updated intercept/bias obtained after gathering.

```
b_reduced: [4.82903972]
```

```
w_reduced: [[ 9.38761910e-07  1.20755041e-04  4.28902716e-09  2.69410
293e-08
   2.02552472e-05  2.85863005e-07  1.80639647e-06  1.01899414e-07
   6.56314888e-10  6.21590026e-08  8.58149506e-10  2.05049800e-05
   1.36117968e-07  8.75017216e-05  2.55818980e-04 -2.00976681e-07
   1.80597088e-04  2.49726398e-07  6.17683215e-05  4.18100235e-07
   2.71846412e-05 -1.19443761e-06  2.37565058e-05  4.34465154e-07
   4.13278500e-05  7.49290178e-05  8.89550675e-09  2.44376737e-10
```

Now after we have obtained all the summed up coefficients and intercept, we need to find the global average coefficients and the intercept using which we have to predict the target value and thereafter, calculate the Loss function (RMSE in our case)

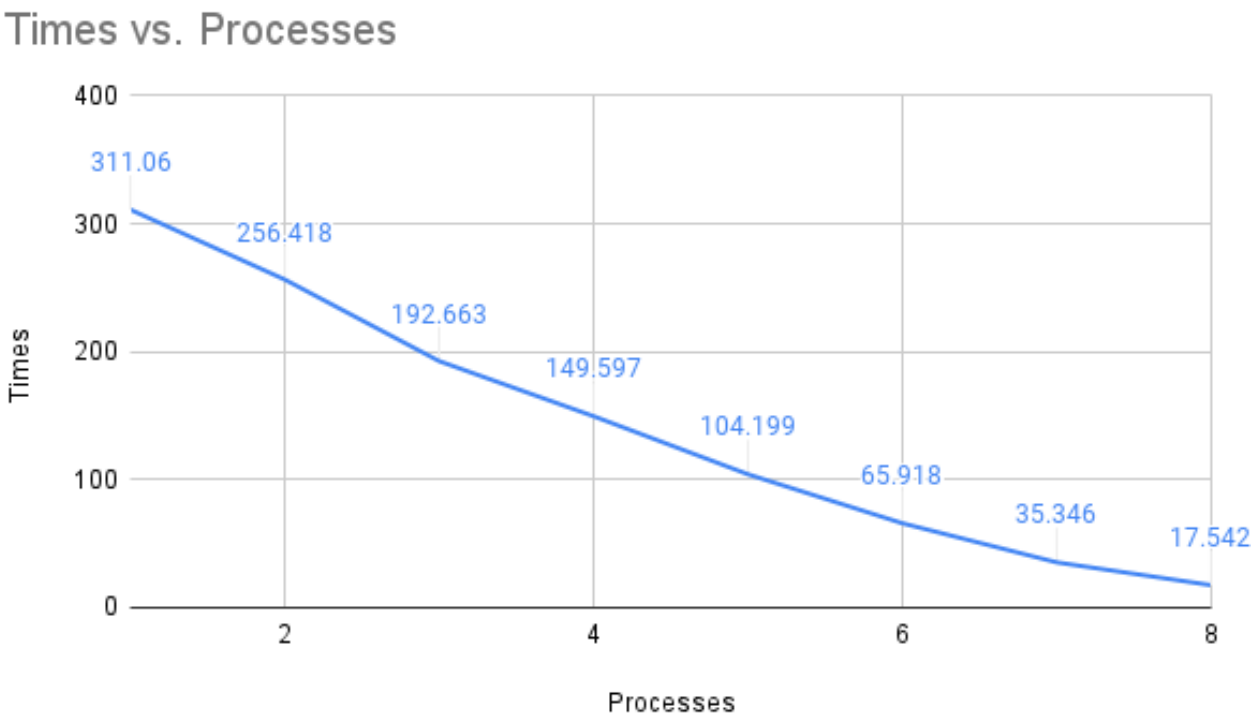
```
if rank == 0:
    times = np.vstack(all_times)
    print('times in Array:', times)
    time_sum = np.sum(times)
    print('Total Time for processes is Net_time=%.3f' % time_sum)
    average_coeffs = w_reduced / size
    average_intercept = b_reduced / size
    print(f'rank:{rank} weights are:{average_coeffs}\n')
    print(f'rank:{rank} coefficient is:{average_intercept}\n')
    y_pred = predict(x_test, average_coeffs, average_intercept)
    print('Mean Squared Error :', RootMeanSquare_Result(y_test, y_pred))
```

The RMSE for different ranks come out to be the same with minimal difference.

The graph of Time taken to compute RMSE vs the number of ranks is given as below:

It is clear that our system is efficient as the time decreases as we increase the number of processes.

Below I have plot the RMSE vs No of processes for P=1,2,3,4,5,6,7,8 and it can be



seen that our program converges somewhere between 125 to 150 epochs, with a learning rate of 0.01.

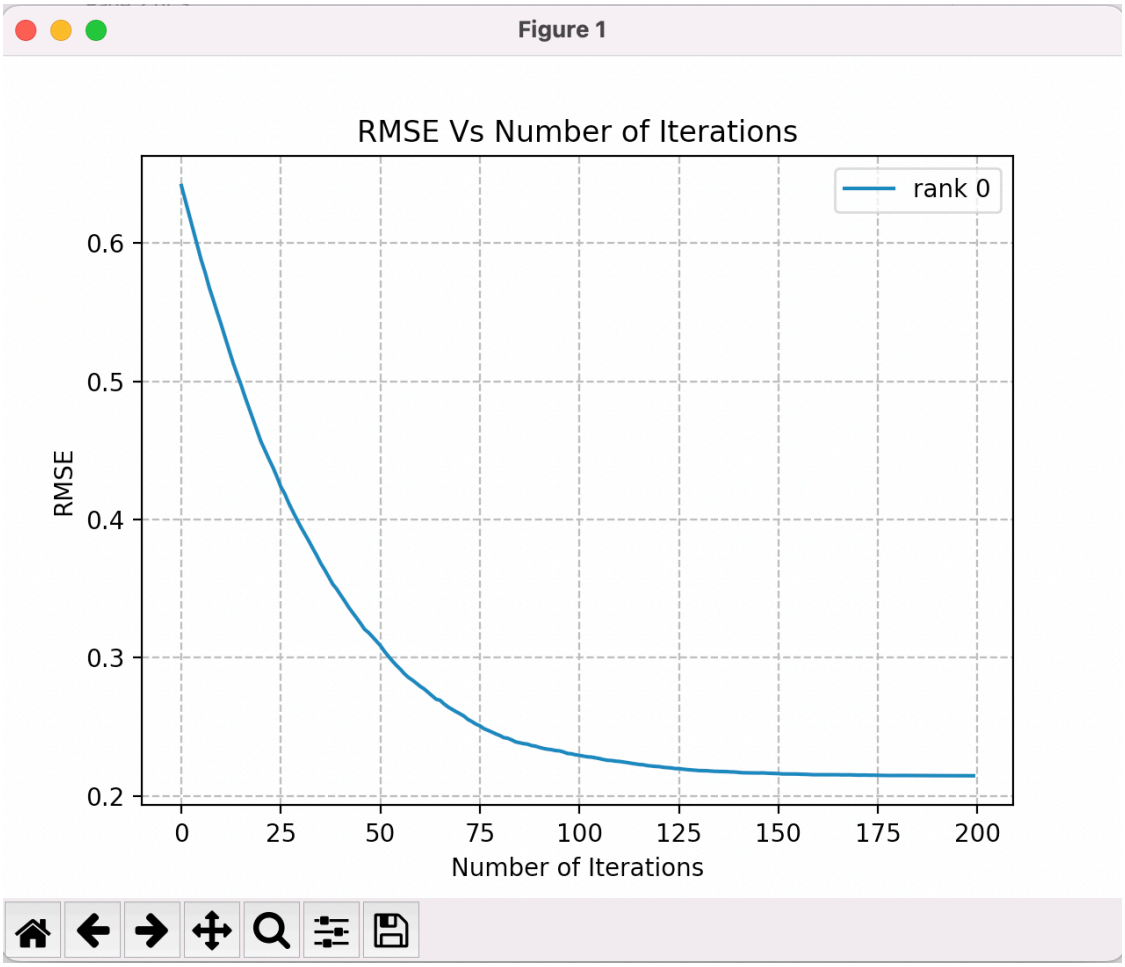


Figure 1
RMSE Vs Number of Iterations
rank 0
RMSE
0.6
0.5
0.4
0.3
0.2
0
25
50
75
100
125
150
175
200
Number of Iterations

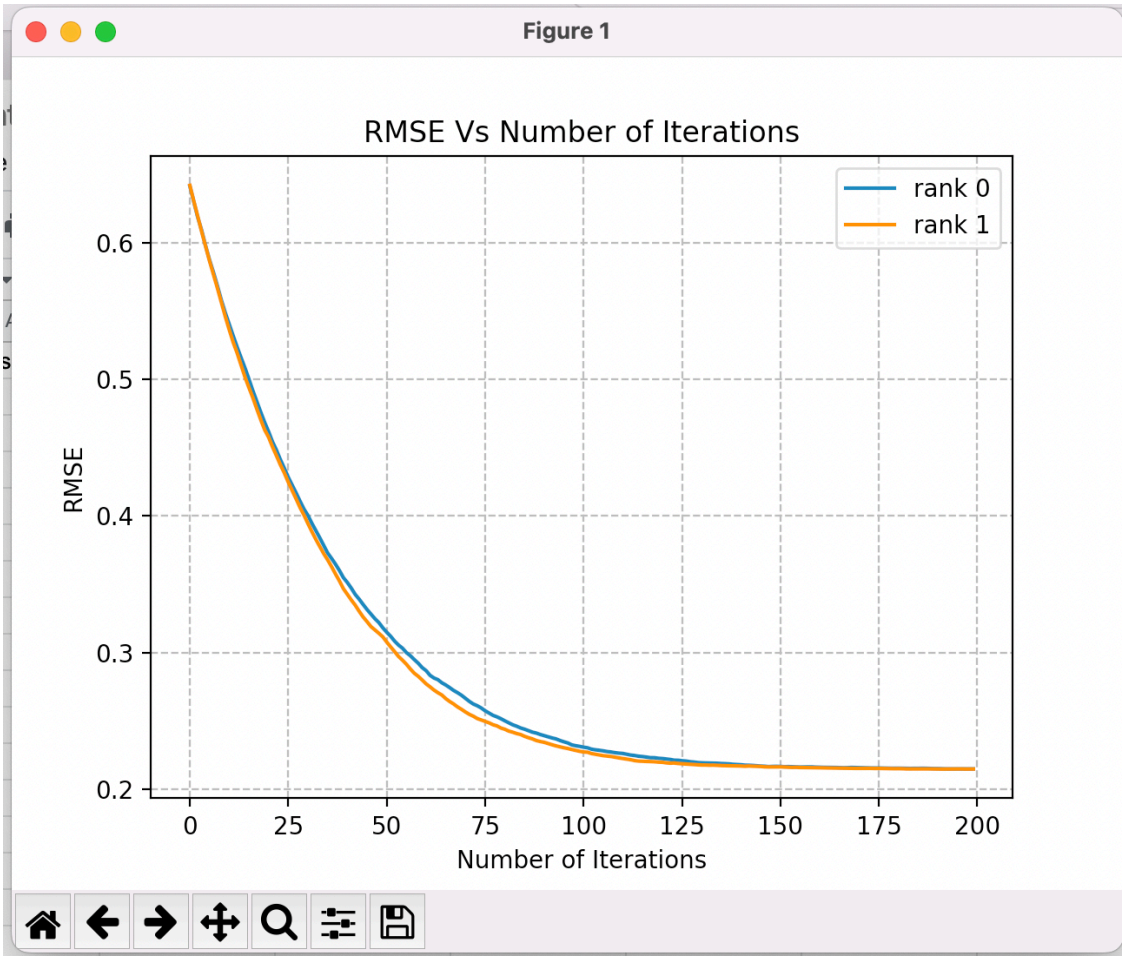


Figure 1
RMSE Vs Number of Iterations
rank 0
rank 1
RMSE
0.6
0.5
0.4
0.3
0.2
0
25
50
75
100
125
150
175
200
Number of Iterations

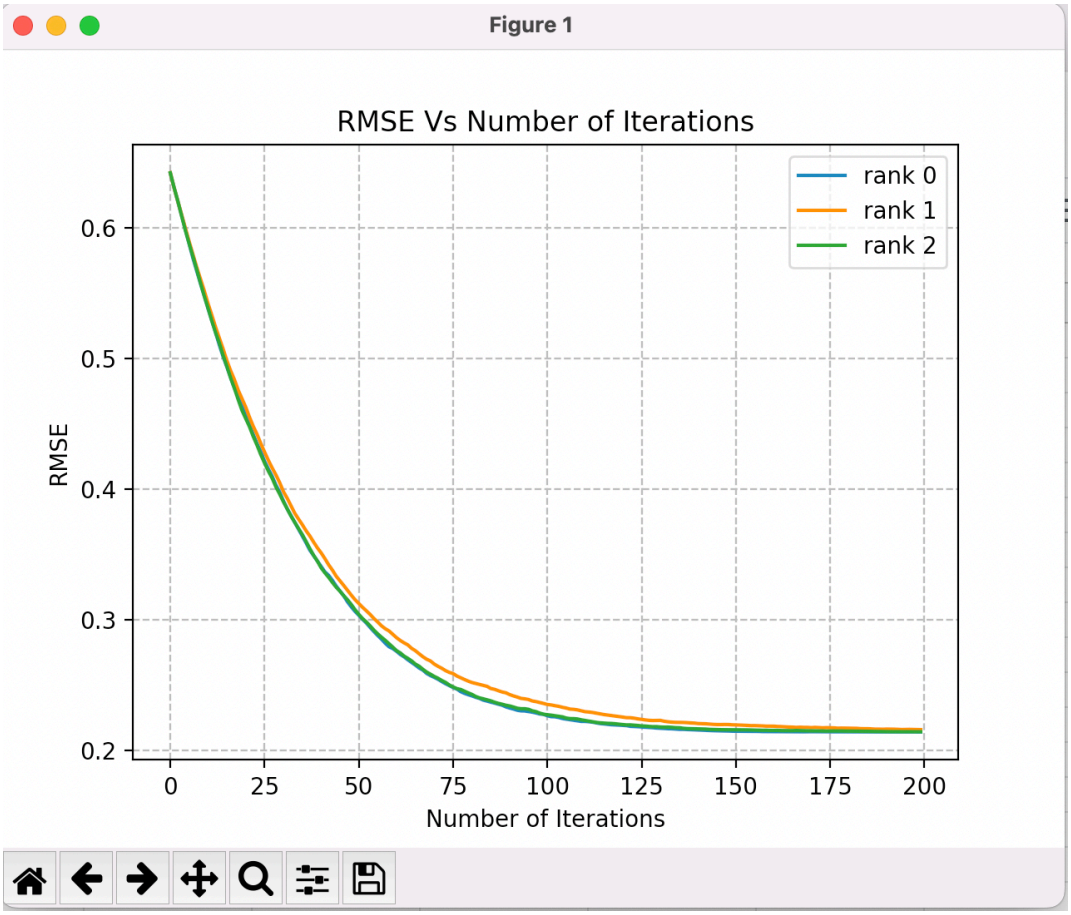


Figure 1
RMSE Vs Number of Iterations
rank 0
rank 1
rank 2
RMSE
0.6
0.5
0.4
0.3
0.2
0
25
50
75
100
125
150
175
200
Number of Iterations

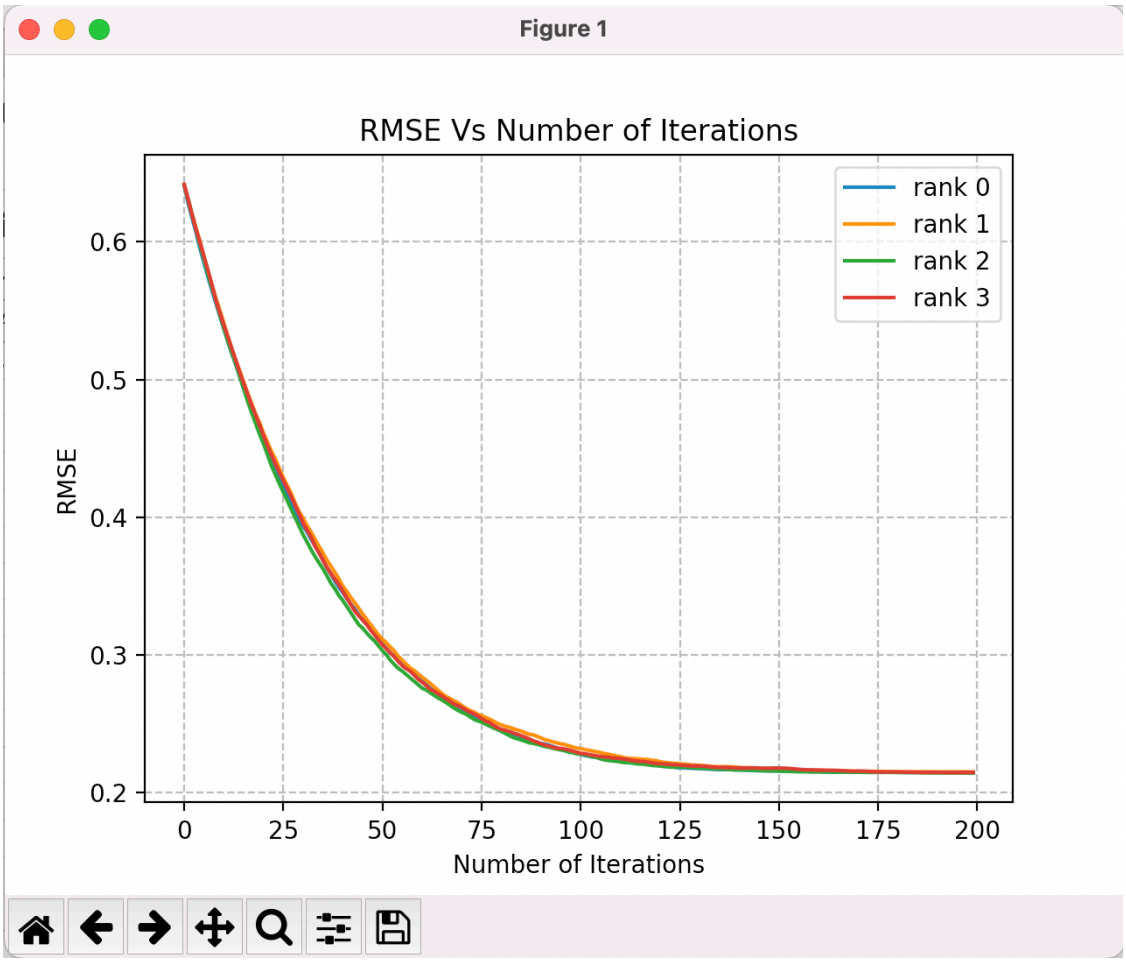


Figure 1
RMSE Vs Number of Iterations
rank 0
rank 1
rank 2
rank 3
RMSE
0.6
0.5
0.4
0.3
0.2
0
25
50
75
100
125
150
175
200
Number of Iterations

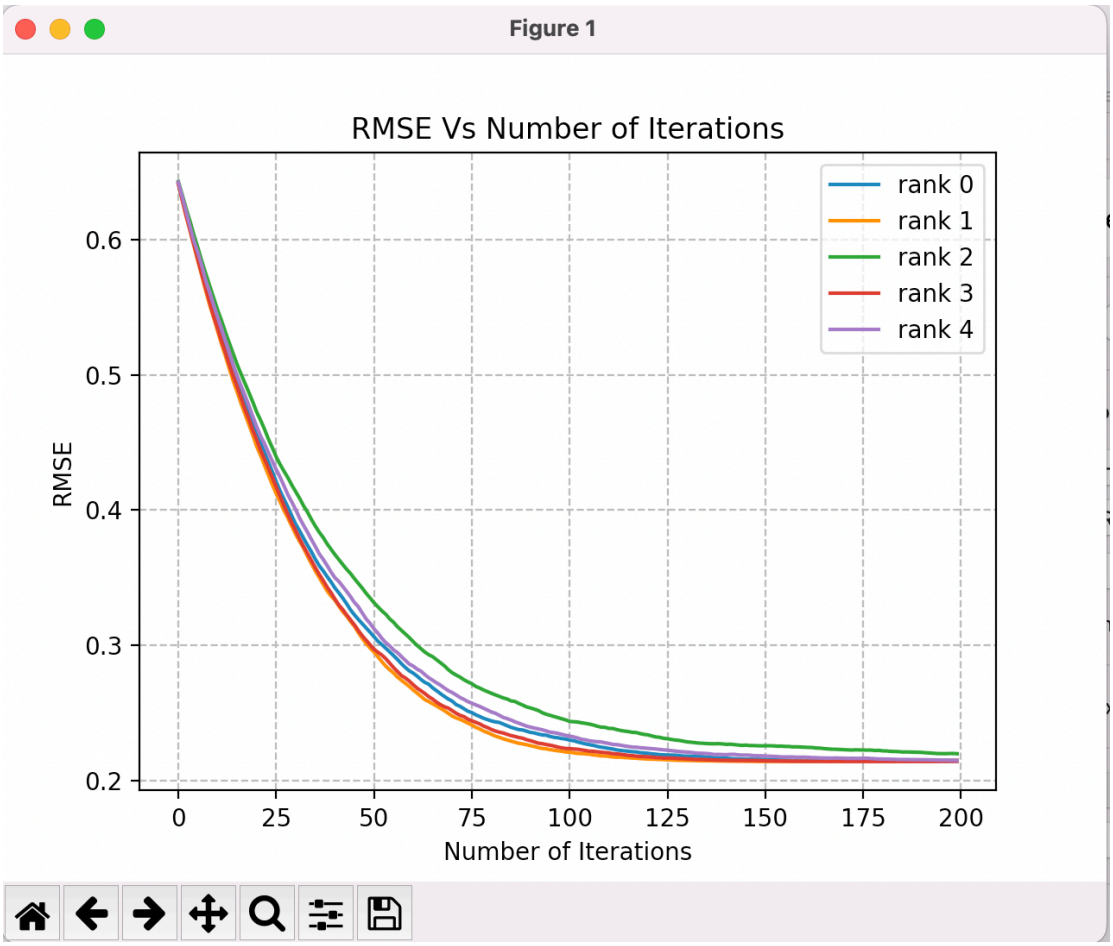


Figure 1
RMSE Vs Number of Iterations
rank 0
rank 1
rank 2
rank 3
rank 4
RMSE
0.2
0.3
0.4
0.5
0.6
0
25
50
75
100
125
150
175
200
Number of Iterations

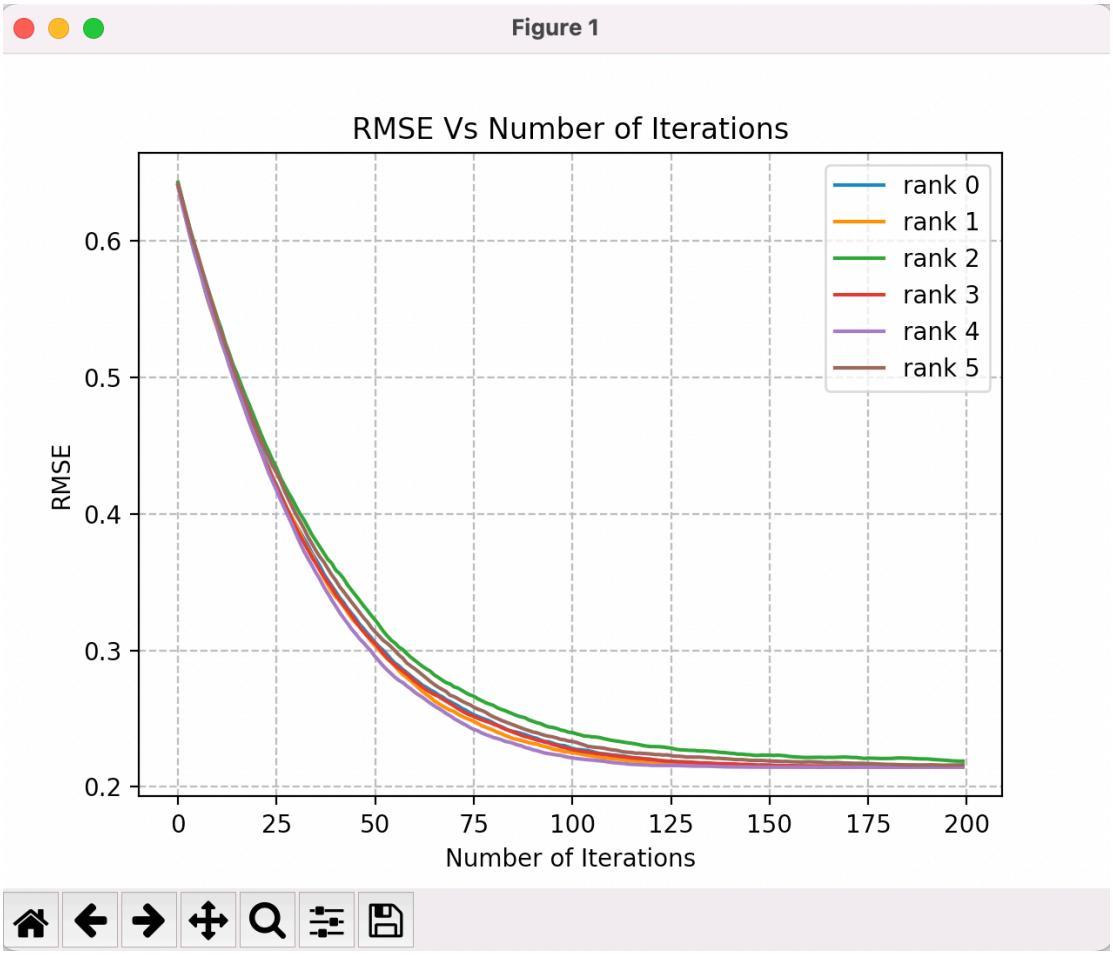


Figure 1
RMSE Vs Number of Iterations
rank 0
rank 1
rank 2
rank 3
rank 4
rank 5
RMSE
0.2
0.3
0.4
0.5
0.6
0
25
50
75
100
125
150
175
200
Number of Iterations

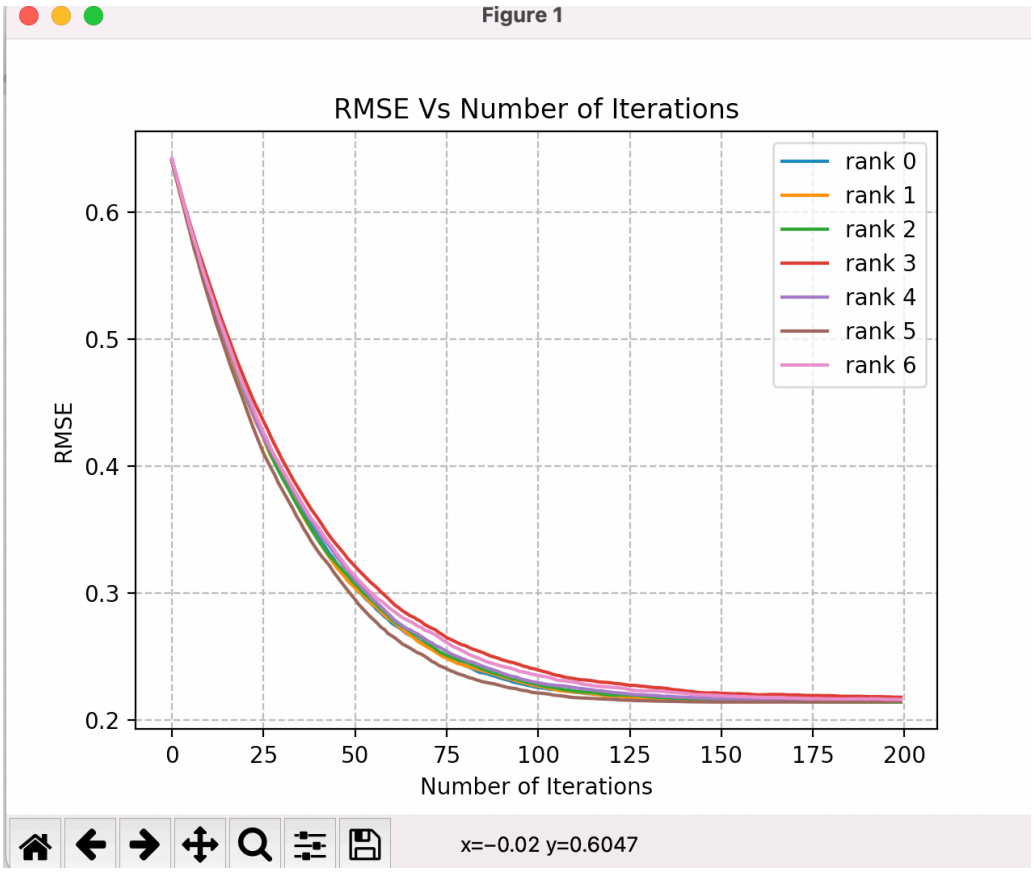


Figure 1
RMSE Vs Number of Iterations
rank 0
rank 1
rank 2
rank 3
rank 4
rank 5
rank 6
RMSE
0.6
0.5
0.4
0.3
0.2
0
25
50
75
100
125
150
175
200
Number of Iterations
x=−0.02 y=0.6047

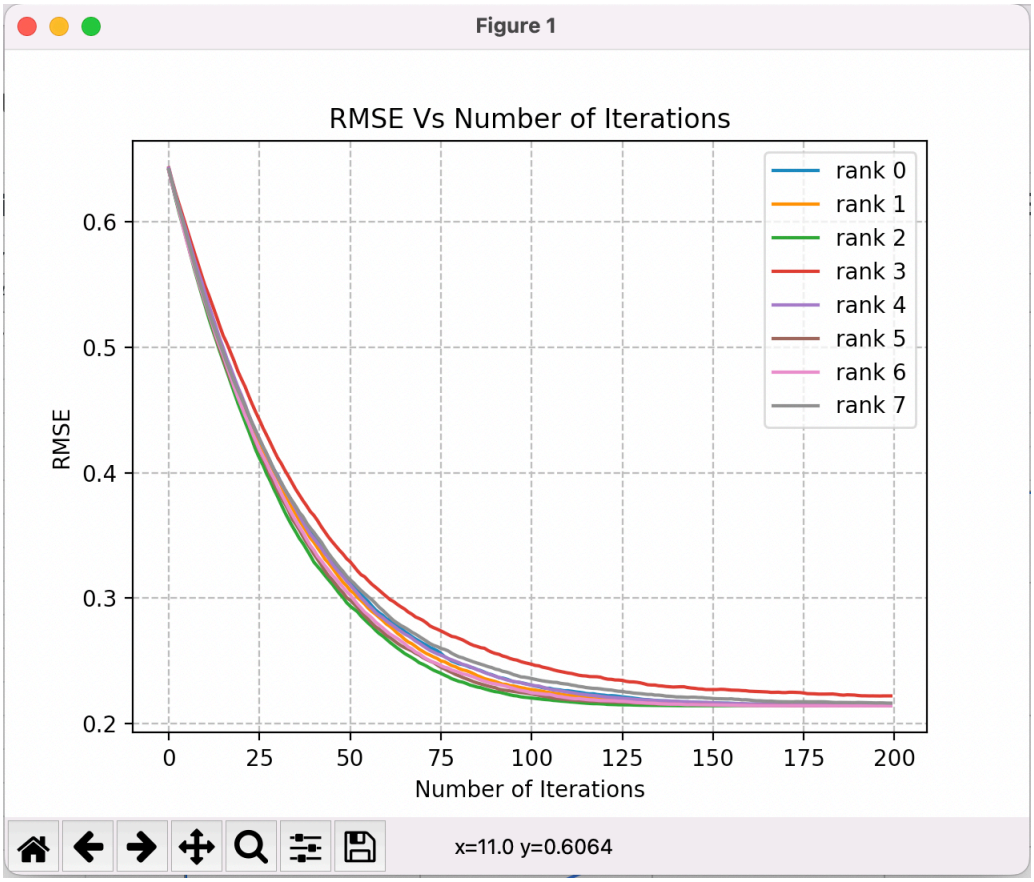


Figure 1
RMSE Vs Number of Iterations
rank 0
rank 1
rank 2
rank 3
rank 4
rank 5
rank 6
rank 7
RMSE
0.6
0.5
0.4
0.3
0.2
0
25
50
75
100
125
150
175
200
Number of Iterations
x=11.0 y=0.6064