University at Buffalo
The State University of New York

CSE574 Introduction to Machine Learning

Project 2

Amir Baghdadi
50135018

Introduction

Learning to Rank (LeToR) is an application of machine learning in the construction of ranking models for information retrieval systems. Training data consists of a lists of items with some partial order specified between items in each list. The order is typically a numerical or ordinal score or a binary judgment for each item. The ranking model's purpose is to rank, i.e. produce a permutation of items in new, unseen lists in a way which is "similar" to rankings in the training data in some sense.

The purpose of this project is to use machine learning approach to solve a LeToR problem. The problem is formulated as a linear regression where the input vector **x** is mapped to a real-valued scalar target $y(\mathbf{x}, \mathbf{w})$. The tasks included training two LeToR and synthetic datasets using closed-form and stochastic gradient descent (SGD) solutions. The data consisted of pairs of input values **x** with different number of features, i.e, 10 for synthetic and 46 for LeToR data, and target value *t*. The input values are real-valued vectors (features derived from a query-document pair) and the target values are scalars (relevance labels) that take one of three values 0, 1, 2, where the larger relevance label corresponds with the better match between query and document. Despite of having discrete targets, linear regression was used to obtain real values that avoids restricting to only three possible values of targets which is more useful in ranking.

Tasks

Extracting feature values and labels from the data:

Features were imported in Python using CSV file read commands. The LeToR data consisted of 69623 data samples and 46 features and the synthetic data consisted of 20000 samples and 10 featurs.

np.shape(letor_input_data) = (69623, 46)

np.shape(syn_input_data) = (20000, 10)

Data partition:

The imported data were partitioned into a training set, validation set, and testing set. The training set takes the first 80% of the data, the validation set is the first 50% of the remaining data (10% of the whole), and the rest is regarded as the testing data. The three sets did not have overlap.

Train model parameter:

The model parameter **w** was trained by a closed-form as well as a Stochastic Gradient Descent (SGD) solution for minimizing the sum-of-squared error between the predicted and the true targets with a regularization term to avoid overfitting.

The closed-form solution has the form

$$\mathbf{w}_{\mathrm{ML}} = (\lambda\mathbf{I} + \mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi})^{-1}\mathbf{\Phi}^{\mathrm{T}}\mathbf{t}\,,$$

where $\mathbf{t} = \{t_1, \ldots, t_N\}$ is the vector of outputs in the training data and $\mathbf{\Phi}$ is the design matrix:

$$\mathbf{\Phi} = \begin{bmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \phi_2(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{bmatrix}$$

The SGD solution takes a random initial value $\mathbf{w}^{(0)}$, and updates the value using

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta^{(\tau)} \nabla E,$$

$$\nabla E = -\left(t_1 - \mathbf{w}^{(\tau)^{\mathrm{T}}} \phi(\mathbf{x}_1)\right) \phi(\mathbf{x}_1) + \lambda \mathbf{w}^{(\tau)}$$

which goes along the opposite direction of the gradient of the error. $\eta^{(\tau)}$ is learning rate that decides how big would be each update step. This process continued for a set number of steps until the convergence occurs. The evaluation of this solution were performed using the Root Mean Squared error

$$E_{RMS} = \sqrt{2E(\mathbf{w}^*)/N}$$

where $\mathbf{w}^*$ is the solution and N is the size of the dataset.

Early stopping rule was also applied here to provide guidance as to how many iterations can be run before the learner begins to over-fit. This early stopping rule work by splitting the original training set into a new training set and a validation set. The error on the validation set is used as a proxy for the generalization error in determining when overfitting has begun. The implementation of early stopping method was as follows:

1. Split the training data into a training set (80%) and a validation set (10%) after shuffling.
2. Train only on the training set and evaluate the per-example error on the validation set once in a while, e.g. after every 10 epoch here known as patience parameter.
3. Stop training as soon as the error on the validation set is higher than it was the last time it was checked.
4. Use the weights the network had in that previous step as the result of the training run.

Tune model hyper-parameter:

The model hyper-parameters included the number of basis functions *M* and the regularization term λ. The values *M* and λ were initially selected by try and error and starting from smaller values and trying some larger values to see the changes in performance known as grid search. In this method, the values of centers for each of the *M* basis function are selected randomly and the spreads were considered uniform for all basis functions, i.e., $(\sigma = 0.5) \times I_{D,\ D=number\ of\ features}$. However, using an advanced method called K-means clustering the value of *M* was selected in a more robust

manner. Here only the results of K-means clustering are reported for the selection of *M*. The learning rate η were selected by a grid search from larger values to smaller ones and different values were considered and the performance in terms of the error rate and convergence time was compared to select the best learning rate value.

K-means clustering:

In order to find the design matrix dimensions in terms of the number of basis functions using a more advanced method, the training dataset was initially clustered using *K*-means clustering algorithm and each cluster was fit with a Gaussian basis function. Different *K* values were compared in terms of the training results and the best value was considered as *M*, i.e., the number of basis functions. The centers for Gaussian radial basis functions, i.e., $\mu_{i,\ i=1,..M}$, were the mean values in each cluster and for each feature (a vector of *1 by 10* for synthetic dataset and a vector of *1 by 46* for LeToR dataset). The spreads for Gaussian radial basis functions, on the other hand, were the covariance matrix functions of the features for each cluster (a matrix of *10 by 10* for synthetic dataset and a matrix of *46 by 46* for LeToR dataset). Six different values of *K* = [1, 2, 3, 4, 5, 6] were considered and the performance of the regression model was compared.

After fixing the tuned model parameters and hyper-parameters, the regression performance was tested on the testing set which shows the ultimate generalization power of the regression gained by learning.

Results:

LeToR dataset:

Considering the validation portion of the data, the hyper-parameters were optimized. The results of K-means clustering show that by changing the *K* value from 1 to 6 the RMS error of the SGD solution slightly reduces, on the other hand this error rate diverges by considering 7 clusters (Fig. 2 and 3). Therefore, the number of clusters were considered to be 6. After fixing the number of clusters (basis functions), λ was tuned by considering different values (λ = [1e-4, 1e-3, 1e-2, 5e-2, 1e-1, 2, 5, 1e+2, 1e+3, 1e+4]) and comparing the performance. The results in Fig. 4 show that the performance reduces by the increasing the λ. The error rate is consistent with smaller λ values and diverges using large ones (>1e+3). 1e-4 was considered here as the optimum value for λ. Analyzing the performance of regression w.r.t learning rate η also shows that the optimum value falls between 0.1 and 0.9 considering η = [0.965, 0.9, 0.5, 0.1, 0.001] for evaluation (Fig. 5). The value of η = 0.5 was considered for the training the regression and evaluating on the testing portion of the dataset.

The results of regression parameters **w** from closed-form and SGD solution are reported here. There are slight difference between closed-form and the SGD solution, however, the plot of RMS error shows that the error rate converges after about 100 iterations (Fig. 1), so there is no need for training the model in more iterations which takes us closer to the closed-form solution results for parameters **w**. It is noteworthy that more number of iterations (100000 iterations) was also examined, however, it has no effect neither on the convergence nor on the overfitting occurrence. In addition, since the results show no overfitting on the validation set, the early stopping algorithms selected the original number of iterations as the final iteration number for the training model. Lastly, considering the parameters **w** from early stopping, i.e., `w_SGD_star`, the performance of the model was assessed on the testing set and the error rate was about 0.56 in terms of the ranking scale (0 to 2) which is equal to about 28%.

Python output:

```
w_closed = [ 0.405  0.702  0.219  0.266  0.018 -0.915 -0.433]
w_SGD = [ 0.19   0.167  0.105  0.133  0.087 -0.334  0.074]
w_SGD_star = [ 0.186  0.163  0.106  0.129  0.087 -0.324  0.076]
early stop iteration number = 100
test error = 0.55 ( 27.63 % )
```

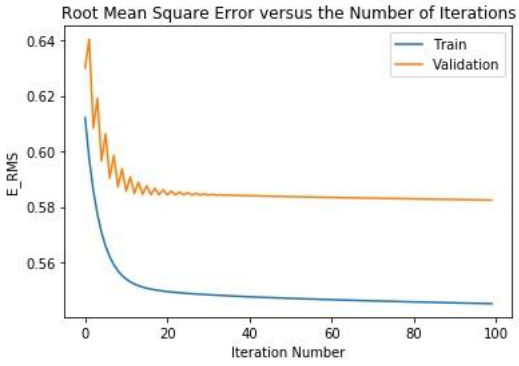

Fig. 1. RMS error of SGD solution

Graphical Results for LeToR dataset:

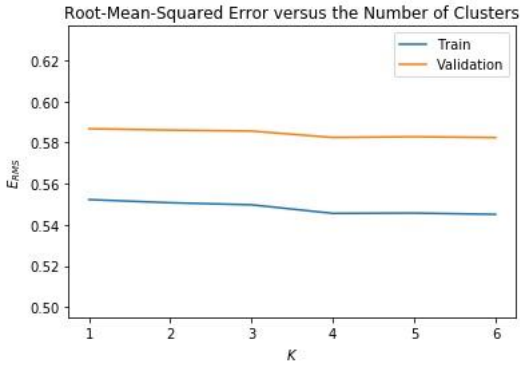

Fig. 2. RMS error of SGD solution change by the number of clusters

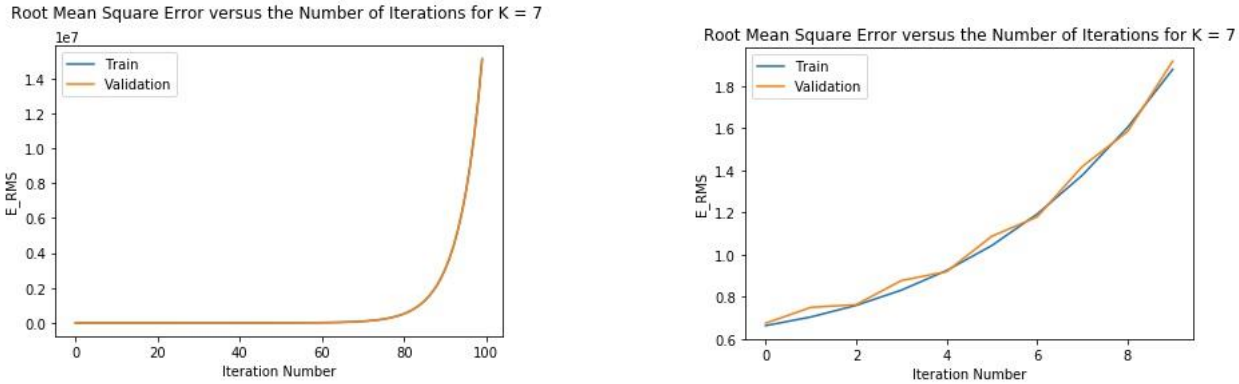

Fig. 3. RMS error of SGD solution for 7 clusters (Left: shown for 100 iteration – Right: shown for 10 iteration)

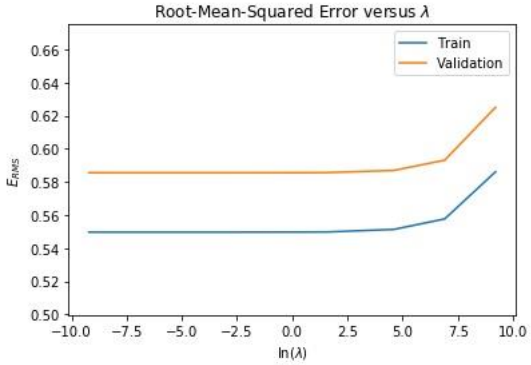

Fig. 4. RMS error of SGD solution change by λ value

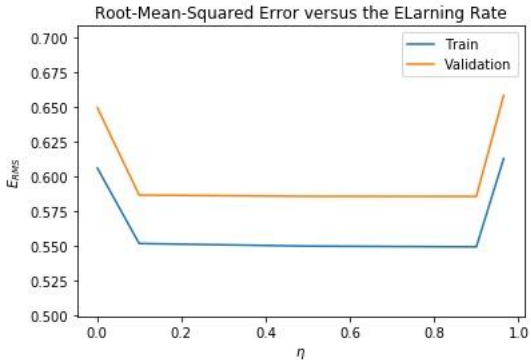

Fig. 5. RMS error of SGD solution change by learning rate value

Synthetic dataset:

The same approach was applied on the synthetic dataset and the values of K = 3, η = 0.4, and λ = 1e-4 was determined as the tuned hyper-parameters. The selected range of hyper-parameters for tuning were as follow:

K = [1, 2, 3]

η = [0.52, 0.4, 0.1, 0.005]

λ = [1e-4, 1e-3, 1e-2, 5e-2, 1e-1, 2, 5, 1e+2]

The same procedure as the LeToR dataset was also applied here and the results of regression parameters **w** is reported.

Python output:

```
w_closed = [-27.008 -70.123  58.829  40.186]
w_SGD = [ 0.189  0.249  0.283  0.269]
w_SGD_star = [ 0.191  0.249  0.282  0.268]
early stop iteration number = 100
test error = 0.77 ( 38.47 % )
```

It is noteworthy that the validation error in synthetic dataset was less than the training error which seems a bit odd in regression analysis. The reason for this behavior can be attributed to the method of data partitioning, so different partitions of data was considered for training and validation sets, however, with no change in the results. This behavior can be described by the nature of the synthetic dataset where the data is generated using a function despite the real data in LeToR dataset which affects the behavior of the regression model.

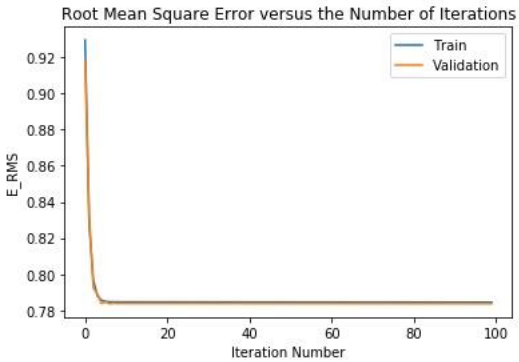

Fig. 6. RMS error of SGD solution

Graphical Results for synthetic dataset:

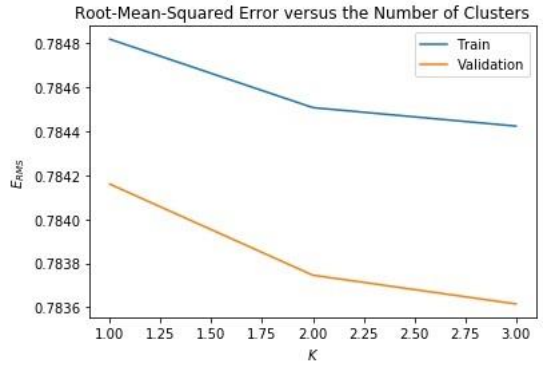

Fig. 7. RMS error of SGD solution change by the number of clusters

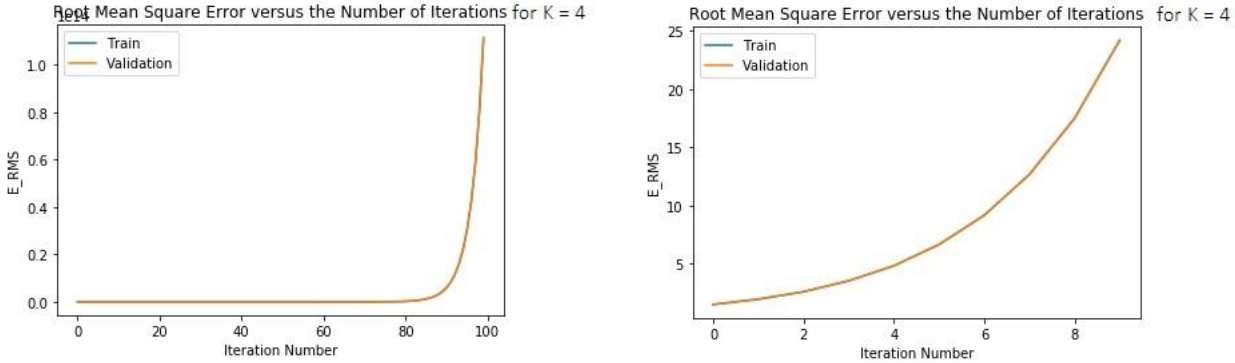

Fig. 8. RMS error of SGD solution for 7 clusters (Left: shown for 100 iteration – Right: shown for 10 iteration)

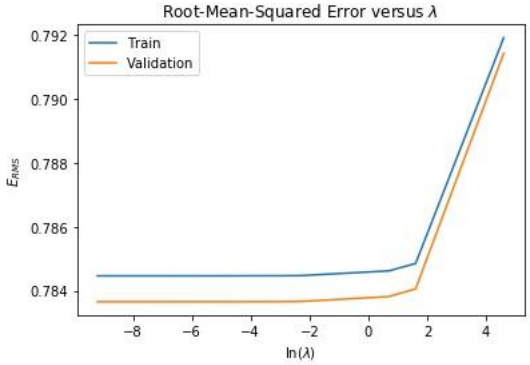

Fig. 9. RMS error of SGD solution change by λ value

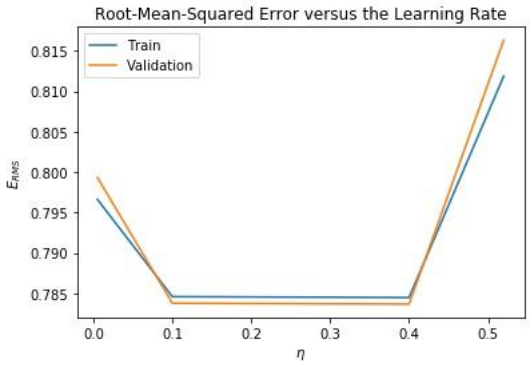

Fig. 10. RMS error of SGD solution change by learning rate value

Project2_forReport.py

```
# Appendix
# Python Code

# Project 2

import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import time
import copy


print('UBitName = amirbagh')
print('personNumber = 50135018')

syn_input_data = pd.read_csv('input.csv',header=None).values
syn_output_data = pd.read_csv('output.csv',header=None).values
letor_input_data = pd.read_csv('Querylevelnorm_X.csv',header=None).values
letor_output_data = pd.read_csv('Querylevelnorm_t.csv',header=None).values

# LeToR or synthetic Data?
data_choose_in = letor_input_data
data_choose_out = letor_output_data
#data_choose_in = syn_input_data
#data_choose_out = syn_output_data

# define hyper-parameters

num_epochs = 100
# 3 for syn and 6 for LeTor
cluster_numbers = 6
# 0.4 for syn and 0.5 for LeToR
learning_rate = 0.5
L2_lambda = 1e-4


def Kmeans_clustring(k, data):
    n_clusters = k
```

```
    n_features = np.shape(data)[1]
    X_cluster = data
    kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(X_cluster)
    labels = kmeans.labels_
    #print(kmeans.predict([[0, 0], [4, 4]]))
    cluster_centroids = kmeans.cluster_centers_
    cluster_spreads = np.empty((n_clusters,n_features,n_features))
    for i in range(k):
        cluster_spreads[i,:,:] = np.cov(X_cluster[np.where(i ==
kmeans.labels_)[0],:].T)

    return (k, cluster_centroids, cluster_spreads)

def dataset_partition(train_percent, val_percent, test_percent, data,
indexes):
    data_lngth = np.shape(data)[0]
    train_data = data[indexes[0:round(train_percent * data_lngth)],:]
    val_data = data[indexes[round(train_percent *
data_lngth):round((train_percent + val_percent) * data_lngth)],:]
    test_data = data[indexes[round((train_percent + val_percent) *
data_lngth):data_lngth],:]
    return (train_data, val_data, test_data)

def compute_design_matrix(input_data, clusters_data):
    # Design matrix
    N, D = input_data.shape
    # shape = [M, 1, D]
    #centers = np.array([np.ones((D))*1, np.ones((D))*0.5, np.ones((D))*1.5])
    centers = clusters_data[1]
    centers = centers[:, np.newaxis, :]
    # shape = [M, D, D]
    #spreads = np.array([np.identity(D), np.identity(D), np.identity(D)]) *
0.5
    spreads = clusters_data[2]
    # shape = [1, N, D]
    X = input_data[np.newaxis, :, :]

    # use broadcast
    basis_func_outputs = np.exp(
```

```
        np.sum(
            np.matmul(X - centers, spreads) * (X - centers),
            axis=2
        ) / (-2)
    ).T
    # insert ones to the 1st col

    return np.insert(basis_func_outputs, 0, 1, axis=1)

def closed_form_sol(L2_lambda, design_matrix, output_data_train):
    return np.linalg.solve(
        L2_lambda * np.identity(design_matrix.shape[1]) +
            np.matmul(design_matrix.T, design_matrix),
        np.matmul(design_matrix.T, output_data_train)
    ).flatten()

def SGD_sol(learning_rate,
            minibatch_size,
            num_epochs,
            L2_lambda,
            design_matrix_train,
            design_matrix_eval,
            output_data_train,
            output_data_eval,
            M):
    N, _ = design_matrix_train.shape
    # You can try different mini-batch size size
    # Using minibatch_size = N is equivalent to standard gradient descent
    # Using minibatch_size = 1 is equivalent to stochastic gradient descent
    # In this case, minibatch_size = N is better
    # The more epochs the higher training accuracy. When set to 1000000,
    # weights will be very close to closed_form_weights. But this is
unnecessary
    weights = np.zeros([1, M+1])
    train_loss = []
    eval_loss = []
    train_err = []
    eval_err = []
    params = np.empty((num_epochs,1,M+1))
```

```
    for epoch in range(num_epochs):
        for i in range(N // minibatch_size):
            lower_bound = i * minibatch_size
            upper_bound = min((i+1)*minibatch_size, N)

            Err_SGD = err_func(Phi = design_matrix_train[lower_bound :
upper_bound, :],
                               weights = weights,
                               t = output_data_train[lower_bound :
upper_bound, :],
                               L2_lambda = L2_lambda,
                               minibatch_size = minibatch_size)
            weights = weights - learning_rate * Err_SGD[3]

            Err_train = err_func(Phi = design_matrix_train[lower_bound :
upper_bound, :],
                                 weights = weights,
                                 t = output_data_train[lower_bound :
upper_bound, :],
                                 L2_lambda = L2_lambda,
                                 minibatch_size = minibatch_size)

            Err_eval = err_func(Phi = design_matrix_eval[lower_bound :
upper_bound, :],
                                weights = weights,
                                t = output_data_eval[lower_bound :
upper_bound, :],
                                L2_lambda = L2_lambda,
                                minibatch_size = minibatch_size)

            train_E =
errorFrom(designMatrix=design_matrix_train,weights=weights.flatten(),targets=o
utput_data_train)

            eval_E =
errorFrom(designMatrix=design_matrix_eval,weights=weights.flatten(),targets=ou
tput_data_eval)

        params[epoch,:,:] = weights
```

```
        train_loss = np.append(train_loss, Err_train[1])
        eval_loss = np.append(eval_loss, Err_eval[1])
        train_err = np.append(train_err, train_E)
        eval_err = np.append(eval_err, eval_E)

    return (params, train_loss, eval_loss, train_err, eval_err)

def err_func(Phi,
             weights,
             t,
             L2_lambda,
             minibatch_size):

    E_D_p = np.sum(np.power(np.matmul(Phi, weights.T)-t, 2)
                           )
    E_D = (E_D_p + L2_lambda * np.matmul(weights, weights.T)) /
(2*minibatch_size)
    E_RMS = np.sqrt(2*E_D)
    #print('E_D =',E_D)
    d_E_p = np.matmul(
            (np.matmul(Phi, weights.T)-t).T,
                     Phi
                    )
    d_E = (d_E_p + L2_lambda * weights) / minibatch_size
    d_E_norm = np.linalg.norm(d_E)
    #print('d_E =',d_E_norm)

    return(E_D, E_RMS, d_E_p, d_E, d_E_norm)


def errorFrom(designMatrix, weights, targets):
    predicted_target = np.dot(designMatrix, weights)
    diff = predicted_target - targets[:,0]
    error = np.sqrt((np.dot(diff.T, diff).item())/diff.shape[0])
    return error

def earlyStopping(max_epochs, patience, n_epochs, params,
this_validation_loss):
    # early-stopping parameters
```

```
    p = patience  # look as this many examples regardless
    theta = np.random.rand(1,4)
    i = 0
    j = 0
    v = np.inf
    theta_s = theta
    i_s = i
    done_looping = False
    while (j < p) and (not done_looping):
        theta = params[i,:,:]
        v_p = this_validation_loss[i]
        i = i + n_epochs
        if v_p < v:
            j = 0
            theta_s = theta
            i_s = i
            v = v_p
        else:
            j = j + 1

        if i >= max_epochs:
            done_looping = True
            break

    return (theta_s, i_s)


# data partition and preprocessing

# shuffling the data
data_index = list(range(0, np.shape(data_choose_in)[0]))
np.random.shuffle(data_index)

data_partition_in = dataset_partition(train_percent=0.8, val_percent=0.1,
test_percent=0.1, data=data_choose_in, indexes=data_index)
data_partition_out = dataset_partition(train_percent=0.8, val_percent=0.1,
test_percent=0.1, data=data_choose_out, indexes=data_index)

input_data_train = data_partition_in[0]
```

```
output_data_train = data_partition_out[0]

input_data_eval = data_partition_in[1]
output_data_eval = data_partition_out[1]

input_data_test = data_partition_in[2]
output_data_test = data_partition_out[2]

clusters_info = Kmeans_clustring(k=cluster_numbers, data=input_data_train)

design_matrix_train = compute_design_matrix(input_data_train, clusters_info)
design_matrix_eval = compute_design_matrix(input_data_eval, clusters_info)
design_matrix_test = compute_design_matrix(input_data_test, clusters_info)

# Closed-form solution
w_closed = closed_form_sol(L2_lambda=L2_lambda,
                           design_matrix=design_matrix_train,
                           output_data_train=output_data_train)
print('w_closed =', np.round(w_closed.flatten(),3))

# Gradient descent solution
SGD_res = SGD_sol(learning_rate=learning_rate,
                  minibatch_size=len(input_data_train),
                  num_epochs=num_epochs,
                  L2_lambda=L2_lambda,
                  design_matrix_train=design_matrix_train,
                  design_matrix_eval=design_matrix_eval,
                  output_data_train=output_data_train,
                  output_data_eval=output_data_eval,
                  M=clusters_info[0])
w_SGD = SGD_res[0]
print('w_SGD =', np.round(w_SGD[-1,:,:].flatten(),3))


# early stop
stop_params = earlyStopping(max_epochs=num_epochs,
                            patience=10,
                            n_epochs=5,
                            params=SGD_res[0],
```

```
                            this_validation_loss=SGD_res[2])

w_SGD_star = stop_params[0].flatten()
print('w_SGD_star =', np.round(stop_params[0].flatten(),3))

early_stop_epoch = stop_params[1]
print('early stop iteration number =', stop_params[1])

# model test
test_E =
errorFrom(designMatrix=design_matrix_test,weights=w_SGD_star,targets=output_da
ta_test)
print('test error =', np.round(test_E,2), '(', round((test_E/2)*100,2),
'%',')')
```