HW 5 By Sayem Lincoln PID - A54207835

1. (70 points) Complete the linear regression functions (least squares and ridge regression) in the class. Use the code to conduct the following experiments:
(a) (10 points) Randomly generate 30 data points from the sine function, where each data point (x; y) has the form:
x = [x0; x1; x2; : : : ; x10]; x 2 [0; 2_]
y = sin(x) + "; " 2 N(0; 0:3)
(b) (10 points) Plot the data points alone with the sine function.
(c) (20 points) Randomly split the dataset (you can use the function provided in the Regres-sion code) and use 30% of the data points for training and the rest for testing. Apply ridge regression using di_erent _ 2 _ = [1e ⍰ 10; 1e ⍰ 5; 1e ⍰ 2; 1e ⍰ 1; 1; 10; 100; 1000].
Plot the training and testing performance.
(d) (30 points) Implement s-fold cross validation. Use k = 4 to choose the optimal _ from the set _ above.

Solution

```
def generate_sin_data(sample_size):
    """
    Generate data with sin function.

    :param sample_size: size of data
    :return: tuple (np.array, np.array)
    """
    # generate X
    X = np.linspace(0, 2 * np.pi, sample_size)
    # generate y
    y = np.sin(X) + np.random.normal(loc=0, scale=0.3, size=sample_size)
    return X.reshape(-1, 1), y.reshape(-1, 1)

def ridge_regression(feature, target, lam=1e-17):
    """
    Compute ridge regression using closed form
    :param feature: X
    :param target: y
    :param lam: lambda
    :return: parameters (np.array)
    """
    feature_dim = feature.shape[1]

    # TODO: Compute the model of ridge regression.
    # closed form solution w=((λI+X.T@X)^−1)@X.T@y
    w = np.linalg.pinv(np.dot(feature.T, feature) + lam *
np.identity(feature_dim)).dot(feature.T.dot(target))
    return w

def k_fold(X, y, k):
    """
    Divide data on k samples.
    :param X: np.array(n, m)
    :param y: np.array(1, n)
    :param k: number of samples to return (int)
    :return: list of tuples of 4 np.arrays
    """
    # create list for all samples
    samples = []
    # get indexes and shuffle
    indexes = np.array(list(range(len(X))))
    # shuffle data
    np.random.shuffle(indexes)
    # get size of test set
    test_size = len(indexes) // k
    i = 0
    counter = 0
    # make folds
    while i < len(indexes):
        # get indexes for train and test sets
        if counter == k-1:
            test_idx = indexes[i: ]
        else:
            test_idx = indexes[i: i + test_size]
        mask = np.ones(len(indexes), bool)
        mask[test_idx] = False
        train_x, train_y = X[mask], y[mask]
        test_x, test_y = X[test_idx], y[test_idx]
        # add sets to sample
        samples.append((train_x, test_x, train_y, test_y))
        i += test_size
        counter += 1
    return samples



def first_part_assignment():
    # generate sin data
    X, y = generate_sin_data(30)
    # create figure and plot
    plt.figure()
    plt.scatter(X, y)
    yh = np.sin(X)
    arr = sorted(zip(X, yh))
    arr_x, arr_y = list(zip(*arr))
    plt.plot(arr_x, arr_y)
    plt.title('Data points & sin function')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.show()
    # devide data on train and test sets
    X_train, X_test, y_train, y_test = rand_split_train_test(X, y, 0.3)
```

```
# create list of lambdas
lambdas = [0.0000000001, 0.00001, 0.01, 0.1, 1, 10, 100, 1000]
# create lists for train and test errors
train_errors = []
test_errors = []
# for each lambda
for lmb in lambdas:
  # compute parameters for ridge regression
  w = ridge_regression(X_train, y_train, lmb)
  # make prediction for train set and compute mse
  y_train_predicted = X_train.dot(w)
  train_errors.append(mean_squared_error(y_train, y_train_predicted))
  # make prediction for test set and compute mse
  y_test_predicted = X_test.dot(w)
  test_errors.append(mean_squared_error(y_test, y_test_predicted))
plt.figure()
plt.plot(lambdas, train_errors, label='Train MSE')
plt.plot(lambdas, test_errors, label='Test MSE')
plt.title('MSE for train and test with different lambdas')
plt.xlabel('Lambdas')
plt.ylabel('MSE')
plt.legend()
plt.show()
# devide data on 4 folds
samples = k_fold(X, y, 4)
# create list for errors
errors = []
# for each lambda
for lmb in lambdas:
  mean_test = []
  # for each fold
  # make regression (find parameters, compute test error)
  for sample in samples:
    X_train_sample, X_test_sample, y_train_sample, y_test_sample = sample
    w = ridge_regression(X_train_sample, y_train_sample, lmb)
    y_test_predicted = X_test_sample.dot(w)
    mean_test.append(mean_squared_error(y_test_sample, y_test_predicted))
  # add mean error for every lambda to list
  errors.append(np.mean(mean_test))
# get lambda with minimum test error
print('The best choice of lambda: {}'.format(lambdas[np.argmin(errors)]))
```

2. (30 points) Complete the gradient descent optimizers (gradient descent and stochastic gradient descent). Use the code to conduct the following experiments:
(a) (10 points) Implement ridge regression using the gradient descent optimizer. Apply gradient descent based ridge regression using a random data set with $N = 1000$ and $d = 50$, and plot the objective changes at each iteration.
(b) (20 points) Apply stochastic gradient descent using a batch size $n = 5; 10; 100; 500$ and for each batch size, plot the objective changes at each iteration.

Solution

```
def get_batches(X, Y, size):
    """
    Divide data into list of mini-batches.

    :param X: np.array(n, m)
    :param y: np.array(1, n)
    :param size: size of mini-batch (int)
    :return: list of tuples (np.array(n, size), np.arrays(1, size))
    """
    # get length of X (and y) - the same
    n_samples = X.shape[0]
    # shuffle data
    indexes = np.array(list(range(X.shape[0])))
    np.random.shuffle(indexes)
    x = X[indexes]
    y = Y[indexes]
    # create list for mini-batches
    samples = []
    # for i from 0 to length of dataset with step of size of mini-batch
    for i in range(0, n_samples, size):
        # find begin and end index of future mini-batch
        begin, end = int(i), int(min(i + size, n_samples))
        # get X and y mini-batches and add them into list
        samples.append((x[begin:end, :], y[begin:end]))
    return samples


def ridge_regression_GD(X, y, lmbda=0.01, l_rate=0.1, tol=0.000001):
    """
    Ridge stochastic gradient descent.

    :param X: np.array(n, m)
    :param y: np.array(1, n)
    :param lmbda: parameter for penalty of 2 (float or int)
    :param l_rate: step of GD (float or int)
    :param tol: tolerance (criteria for stopping)
    :return: tuple(np.array(X.shape[0], 1), list of floats)
    """

    # initialize w: parameters of the model
    w = np.zeros((X.shape[1], 1))
    # create list for losses
    errors = []
    converged = False
    i = 1
    old_error = 0
    # while algorithm does not converge
    while not converged:
        # get prediction
        yhat = X.dot(w)
        # start to compute gradient
        # get error
        error = yhat - y
        # update parameters
        w -= l_rate * (X.T.dot(error) + lmbda * w) / X.shape[0]
        # get loss
        new_error = (np.sum((y - X.dot(w))**2) + lmbda * np.linalg.norm(w))/ (2 * X.shape[0])
        # check how loss from last and current iteration changes
        # if difference between current and previous loss is less then tolerance
        # algorithm converged
        if np.abs(old_error - new_error) < tol:
            converged = True
        # increase number of iteration
        i += 1
        # save current loss
        errors.append(new_error)
        # update old loss to current
        old_error = new_error
        # print loss if iteration number divides by 100
        if i % 100 == 0:
            print(i, new_error)
    return w, errors

def ridge_regression_SGD(X, Y, batch, lmbda=0.01, l_rate=0.01, tol=0.0000001):
    """
    Ridge regression stochastic gradient descent.

    :param X: np.array(n, m)
    :param y: np.array(1, n)
    :param batch: size of batch (int)
    :param lmbda: parameter for penalty of 2 (float or int)
    :param l_rate: step of GD (float or int)
    :param tol: tolerance (criteria for stopping)
    :return: tuple(np.array(X.shape[0], 1), list of floats)
    """
    # initialize w: parameters of the model
    w = np.zeros((X.shape[1], 1))
    # create list for losses
    errors = []
    # devide data on mini-batches
    batches = get_batches(X, Y, batch)
    converged = False
    i = 1
    old_error = 0
    # while algorithm does not converge
    while not converged:
        # create list for store loss on each batch
        batch_errors = []
        # for every part of X and y
        for (x, y) in batches:
            # get prediction
            yhat = x.dot(w)
            # get error
            error = yhat - y
            # update parameter
            w -= l_rate * (x.T.dot(error) + lmbda * w) / X.shape[0]
            # compute loss on current batch
            batch_error = (np.sum((y - x.dot(w))**2) + lmbda * np.linalg.norm(w))/ (2 *
X.shape[0] * i)
            batch_errors.append(batch_error)
            # add to list of all losses
            errors.append(batch_error)
        # compute mean loss of current iteration
        new_error = np.mean(batch_errors)
        # check how loss from last and current iteration changes
        # if difference between current and previous loss is less then tolerance
        # algorithm converged
        if np.abs(old_error - new_error) < tol:
            converged = True
        # increase number of current iteration
        i += 1
        # update current loss
        old_error = new_error
        if i % 100 == 0:
            print(i, new_error)
    return w, errors

def second_part_assignment():
    # generate random data with 1000 samples and 30 features
    X = np.random.randn(50).reshape(1, -1)
    for i in range(999):
        X = np.vstack((X, np.random.randn(50).reshape(1, -1)))
    y = np.random.randn(1000) * np.random.randn(1000)
    y = y.reshape(-1, 1)
    # compute model parameters and losses
    w, errors = ridge_regression_GD(X, y, l_rate=0.01, lmbda=0.01, tol=0.00000001)
    # plot results
    plt.figure()
    plt.plot(list(range(len(errors))), errors)
    plt.xlabel('Number of iterations')
    plt.ylabel('Loss')
    plt.title('Gradient descent ridge regression loss')
    plt.show()
    print('Got MSE: {}'.format(mean_squared_error(y, X.dot(w))))
    # create list for errors
    mean_errors = []
    # for different sizes of batches
    for batch in [5, 10, 100, 500]:
        # create plot
        plt.figure()
        # compute parameters and losses with stochastic GD
        w, errors = ridge_regression_SGD(X, y, lmbda=0.1, l_rate=0.1, batch=batch)
        # compute mse
        mean_errors.append(mean_squared_error(y, X.dot(w)))
        # plot losses
        plt.plot(list(range(len(errors))), errors)
        plt.title('SGD Ridge regression loss with batch size {}'.format(batch))
        plt.xlabel('Number of iterations')
        plt.ylabel('Loss')
        plt.show()
    # create plot for mse of ridge with parameters got while training SGD
    plt.figure()
    plt.plot([5, 10, 100, 500], mean_errors)
    plt.title('MSE')
    plt.xlabel('Batch size')
    plt.ylabel('Loss')
    plt.show()
```

Whole solution in regression_class.py

```
import time
import numpy as np
from sklearn.utils import shuffle
import matplotlib.pyplot as plt


def rand_split_train_test(data, label, train_perc):
    if train_perc >= 1 or train_perc <= 0:
        raise Exception('train_perc should be between (0,1).')
    sample_size = data.shape[0]
    if sample_size < 2:
        raise Exception('Sample size should be larger than 1. ')

    num_train_sample = np.max([np.floor(sample_size * train_perc).astype(int), 1])
    data, label = shuffle(data, label)

    data_tr = data[:num_train_sample]
    data_te = data[num_train_sample:]

    label_tr = label[:num_train_sample]
    label_te = label[num_train_sample:]

    return data_tr, data_te, label_tr, label_te


def subsample_data(data, label, subsample_size):
    # protected sample size
    subsample_size = np.max([1, np.min([data.shape[0], subsample_size])])
    data, label = shuffle(data, label)
    data = data[:subsample_size]
    label = label[:subsample_size]
    return data, label


def generate_rnd_data(feature_size, sample_size, bias=False):

    # Generate X matrix.
    data = np.concatenate((np.random.randn(sample_size, feature_size), np.ones((sample_size,
1))), axis=1) \
        if bias else np.random.randn(sample_size, feature_size)  # the first dimension is sample_size
(n X d)

    # Generate ground truth model.
    truth_model = np.random.randn(feature_size + 1, 1) * 10 \
        if bias else np.random.randn(feature_size, 1) * 10

    # Generate label.
    label = np.dot(data, truth_model)

    # add element-wise gaussian noise to each label.
    label += np.random.randn(sample_size, 1)
    return data, label, truth_model


def mean_squared_error(true_label, predicted_label):
    """
        Compute the mean square error between the true and predicted labels
        :param true_label: Nx1 vector
        :param predicted_label: Nx1 vector
        :return: scalar MSE value
    """
    mse = np.sqrt(np.sum((true_label - predicted_label)**2)/true_label.size)
    return mse

def least_squares(feature, target):
    """
    Compute least squares using closed form
    :param feature: X
    :param target: y
    :return: computed weight vector
    """

    # TODO: Compute the model of least squares.
    w = np.linalg.pinv(np.dot(feature.T, feature)).dot(feature.T.dot(target))
    return w
```

```
def generate_sin_data(sample_size):
    """
    Generate data with sin function.

    :param sample_size: size of data
    :return: tuple (np.array, np.array)
    """
    # generate X
    X = np.linspace(0, 2 * np.pi, sample_size)
    # generate y
    y = np.sin(X) + np.random.normal(loc=0, scale=0.3, size=sample_size)
    return X.reshape(-1, 1), y.reshape(-1, 1)


def ridge_regression(feature, target, lam=1e-17):
    """
    Compute ridge regression using closed form
    :param feature: X
    :param target: y
    :param lam: lambda
    :return: parameters (np.array)
    """
    feature_dim = feature.shape[1]

    # TODO: Compute the model of ridge regression.
    # closed form solution w=((λI+X.T@X)^−1)@X.T@y
    w = np.linalg.pinv(np.dot(feature.T, feature) + lam *
np.identity(feature_dim)).dot(feature.T.dot(target))
    return w

def get_batches(X, Y, size):
    """
    Divide data into list of mini-batches.

    :param X: np.array(n, m)
    :param y: np.array(1, n)
    :param size: size of mini-batch (int)
    :return: list of tuples (np.array(n, size), np.arrays(1, size))
    """
    # get length of X (and y) - the same
    n_samples = X.shape[0]
    # shuffle data
    indexes = np.array(list(range(X.shape[0])))
    np.random.shuffle(indexes)
    x = X[indexes]
    y = Y[indexes]
    # create list for mini-batches
    samples = []
    # for i from 0 to length of dataset with step of size of mini-batch
    for i in range(0, n_samples, size):
        # find begin and end index of future mini-batch
        begin, end = int(i), int(min(i + size, n_samples))
        # get X and y mini-batches and add them into list
        samples.append((x[begin:end, :], y[begin:end]))
    return samples


def ridge_regression_GD(X, y, lmbda=0.01, l_rate=0.1, tol=0.000001):
    """
    Ridge stochastic gradient descent.

    :param X: np.array(n, m)
    :param y: np.array(1, n)
    :param lmbda: parameter for penalty of 2 (float or int)
    :param l_rate: step of GD (float or int)
    :param tol: tolerance (criteria for stopping)
    :return: tuple(np.array(X.shape[0], 1), list of floats)
    """

    # initialize w: parameters of the model
    w = np.zeros((X.shape[1], 1))
    # create list for losses
    errors = []
    converged = False
    i = 1
    old_error = 0
    # while algorithm does not converge
    while not converged:
        # get prediction
        yhat = X.dot(w)
        # start to compute gradient
        # get error
        error = yhat - y
        # update parameters
        w -= l_rate * (X.T.dot(error) + lmbda * w) / X.shape[0]
        # get loss
        new_error = (np.sum((y - X.dot(w))**2) + lmbda * np.linalg.norm(w))/ (2 * X.shape[0])
        # check how loss from last and current iteration changes
        # if difference between current and previous loss is less then tolerance
        # algorithm converged
        if np.abs(old_error - new_error) < tol:
            converged = True
        # increase number of iteration
        i += 1
        # save current loss
        errors.append(new_error)
        # update old loss to current
        old_error = new_error
        # print loss if iteration number divides by 100
        if i % 100 == 0:
```

```python
        print(i, new_error)
    return w, errors

def ridge_regression_SGD(X, Y, batch, lmbda=0.01, l_rate=0.01, tol=0.0000001):
    """
    Ridge regression stochastic gradient descent.

    :param X: np.array(n, m)
    :param y: np.array(1, n)
    :param batch: size of batch (int)
    :param lmbda: parameter for penalty of 2 (float or int)
    :param l_rate: step of GD (float or int)
    :param tol: tolerance (criteria for stopping)
    :return: tuple(np.array(X.shape[0], 1), list of floats)
    """
    # initialize w: parameters of the model
    w = np.zeros((X.shape[1], 1))
    # create list for losses
    errors = []
    # devide data on mini-batches
    batches = get_batches(X, Y, batch)
    converged = False
    i = 1
    old_error = 0
    # while algorithm does not converge
    while not converged:
        # create list for store loss on each batch
        batch_errors = []
        # for every part of X and y
        for (x, y) in batches:
            # get prediction
            yhat = x.dot(w)
            # get error
            error = yhat - y
            # update parameter
            w -= l_rate * (x.T.dot(error) + lmbda * w) / X.shape[0]
            # compute loss on current batch
            batch_error = (np.sum((y - x.dot(w))**2) + lmbda * np.linalg.norm(w))/ (2 * X.shape[0] * i)
            batch_errors.append(batch_error)
            # add to list of all losses
            errors.append(batch_error)
        # compute mean loss of current iteration
        new_error = np.mean(batch_errors)
        # check how loss from last and current iteration changes
        # if difference between current and previous loss is less then tolerance
        # algorithm converged
        if np.abs(old_error - new_error) < tol:
            converged = True
        # increase number of current iteration
        i += 1
        # update current loss
        old_error = new_error
        if i % 100 == 0:
            print(i, new_error)
    return w, errors



def k_fold(X, y, k):
    """
    Divide data on k samples.

    :param X: np.array(n, m)
    :param y: np.array(1, n)
    :param k: number of sumples to return (int)
    :return: list of tuples of 4 np.arrays
    """
    # create list for all samples
    samples = []
    # get indexes and shuffle
    indexes = np.array(list(range(len(X))))
    # shuffle data
    np.random.shuffle(indexes)
    # get size of test set
    test_size = len(indexes) // k
    i = 0
    counter = 0
    # make folds
    while i < len(indexes):
        # get indexes for train and test sets
        if counter == k-1:
            test_idx = indexes[i: ]
        else:
            test_idx = indexes[i: i + test_size]
        mask = np.ones(len(indexes), bool)
        mask[test_idx] = False
        train_x, train_y = X[mask], y[mask]
        test_x, test_y = X[test_idx], y[test_idx]
        # add sets to sample
        samples.append((train_x, test_x, train_y, test_y))
        i += test_size
        counter += 1
    return samples



def exp1():
    # EXP1: training testing.
    # generate a data set.
    (feature_all, target_all, model) = generate_rnd_data(feature_size=3, sample_size=20,
bias=True)
    # split training/testing
    feature_train, feature_test, target_train, target_test = rand_split_train_test(feature_all,
target_all, train_perc=0.8)
    # compute model
    reg_model_lsqr = least_squares(feature_train, target_train)
    reg_model_ridge = ridge_regression(feature_train, target_train, lam=1e-7)

    # evaluate performance
    print('Training MSE(lsqr):', mean_squared_error(target_train, np.dot(feature_train,
reg_model_lsqr)))
    print('Testing MSE(lsqr):', mean_squared_error(target_test, np.dot(feature_test,
reg_model_lsqr)))
    print('Training MSE(ridge):', mean_squared_error(target_train, np.dot(feature_train,
reg_model_ridge)))
    print('Testing MSE(ridge):', mean_squared_error(target_test, np.dot(feature_test,
reg_model_ridge)))


def exp2():
    # EXP2: generalization performance: increase sample size.
    different_sample_sizes = [50, 100, 150, 200, 250, 300, 350, 400, 450]
    (feature_all, target_all, model) = generate_rnd_data(feature_size=100, sample_size=1000,
bias=True)
    feature_hold, feature_test, target_hold, target_test = \
        rand_split_train_test(feature_all, target_all, train_perc=0.9)

    train_performance = []
    test_performance = []
    for train_sample_size in different_sample_sizes:
        feature_train, target_train = subsample_data(feature_hold, target_hold, train_sample_size)
        reg_model = ridge_regression(feature_train, target_train, lam=1e-5)
        train_performance += [mean_squared_error(target_train, np.dot(feature_train, reg_model))]
        test_performance += [mean_squared_error(target_test, np.dot(feature_test, reg_model))]

    print(train_performance)
    print(test_performance)

    plt.figure()
    train_plot, = plt.plot(different_sample_sizes, np.log10(train_performance), linestyle='-',
color='b',
                           label='Training Error')
    test_plot, = plt.plot(different_sample_sizes, np.log10(test_performance), linestyle='-', color='r',
label='Testing '
                                                                              'Error')
    plt.xlabel("Sample Size")
    plt.ylabel("Error (log)")
    plt.title("Generalization performance: increase sample size fix dimensionality")
    plt.legend(handles=[train_plot, test_plot])
    plt.show()
```

```
def exp3():
    # EXP3: generalization performance: increase dimensionality.
    different_dimensionality = [100, 150, 200, 250, 300, 350, 400, 450]

    train_performance = []
    test_performance = []
    for dimension in different_dimensionality:
        (feature_all, target_all, model) = generate_rnd_data(feature_size=dimension,
sample_size=1000, bias=True)
        feature_train, feature_test, target_train, target_test = \
            rand_split_train_test(feature_all, target_all, train_perc=0.9)
        reg_model = ridge_regression(feature_train, target_train, lam=1e-5)
        train_performance += [mean_squared_error(target_train, np.dot(feature_train, reg_model))]
        test_performance += [mean_squared_error(target_test, np.dot(feature_test, reg_model))]

    print(train_performance)
    print(test_performance)

    plt.figure()
    train_plot, = plt.plot(different_dimensionality, np.log10(train_performance), linestyle='-',
color='b',
                          label='Training Error')
    test_plot, = plt.plot(different_dimensionality, np.log10(test_performance), linestyle='-',
color='r', label='Testing '
                                                                          'Error')
    plt.xlabel("Dimensionality")
    plt.ylabel("Error (log)")
    plt.title("Generalization performance: increase dimensionality fix sample size")
    plt.legend(handles=[train_plot, test_plot])
    plt.show()


def exp4():
    # EXP4: computational time: increase dimensionality.
    different_dimensionality = range(100, 2000, 100)

    train_performance = []
    test_performance = []
    time_elapse = []
    for dimension in different_dimensionality:
        (feature_all, target_all, model) = generate_rnd_data(feature_size=dimension,
sample_size=1000, bias=True)
        feature_train, feature_test, target_train, target_test = \
            rand_split_train_test(feature_all, target_all, train_perc=0.9)
        t = time.time()
        reg_model = ridge_regression(feature_train, target_train, lam=1e-5)
        time_elapse += [time.time() - t]
        print('Finished model of dimension {}'.format(dimension))
        train_performance += [mean_squared_error(target_train, np.dot(feature_train, reg_model))]
        test_performance += [mean_squared_error(target_test, np.dot(feature_test, reg_model))]

    plt.figure()
    time_plot, = plt.plot(different_dimensionality, time_elapse, linestyle='-', color='r', label='Time
cost')
    plt.xlabel("Dimensionality")
    plt.ylabel("Time (ms)")
    plt.title("Computational efficiency.")
    plt.legend(handles=[time_plot])
    plt.show()


def first_part_assignment():
    # generate sin data
    X, y = generate_sin_data(30)
    # create figure and plot
    plt.figure()
    plt.scatter(X, y)
    yh = np.sin(X)
    arr = sorted(zip(X, yh))
    arr_x, arr_y = list(zip(*arr))
    plt.plot(arr_x, arr_y)
    plt.title('Data points & sin function')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.show()
    # devide data on train and test sets
    X_train, X_test, y_train, y_test = rand_split_train_test(X, y, 0.3)
    # create list of lambdas
    lambdas = [0.0000000001, 0.00001, 0.01, 0.1, 1, 10, 100, 1000]
    # create lists for train and test errors
    train_errors = []
    test_errors = []
    # for each lambda
    for lmb in lambdas:
        # compute parameters for ridge regression
        w = ridge_regression(X_train, y_train, lmb)
        # make prediction for train set and compute mse
        y_train_predicted = X_train.dot(w)
        train_errors.append(mean_squared_error(y_train, y_train_predicted))
        # make prediction for test set and compute mse
        y_test_predicted = X_test.dot(w)
        test_errors.append(mean_squared_error(y_test, y_test_predicted))
    plt.figure()
    plt.plot(lambdas, train_errors, label='Train MSE')
    plt.plot(lambdas, test_errors, label='Test MSE')
    plt.title('MSE for train and test with different lambdas')
    plt.xlabel('Lambdas')
    plt.ylabel('MSE')
    plt.legend()
    plt.show()
    # devide data on 4 folds
    samples = k_fold(X, y, 4)
    # create list for errors
    errors = []
    # for each lambda
    for lmb in lambdas:
        mean_test = []
        # for each fold
        # make regression (find parameters, compute test error)
        for sample in samples:
            X_train_sample, X_test_sample, y_train_sample, y_test_sample = sample
            w = ridge_regression(X_train_sample, y_train_sample, lmb)
            y_test_predicted = X_test_sample.dot(w)
            mean_test.append(mean_squared_error(y_test_sample, y_test_predicted))
        # add mean error for every lambda to list
        errors.append(np.mean(mean_test))
    # get lambda with minimum test error
    print('The best choice of lambda: {}'.format(lambdas[np.argmin(errors)]))


def second_part_assignment():
    # generate random data with 1000 samples and 30 features
    X = np.random.randn(50).reshape(1, -1)
    for i in range(999):
        X = np.vstack((X, np.random.randn(50).reshape(1, -1)))
    y = np.random.randn(1000) * np.random.randn(1000)
    y = y.reshape(-1, 1)
    # compute model parameters and losses
    w, errors = ridge_regression_GD(X, y, l_rate=0.01, lmbda=0.01, tol=0.00000001)
    # plot results
    plt.figure()
    plt.plot(list(range(len(errors))), errors)
    plt.xlabel('Number of iterations')
    plt.ylabel('Loss')
    plt.title('Gradient descent ridge regression loss')
    plt.show()
    print('Got MSE: {}'.format(mean_squared_error(y, X.dot(w))))
    # create list for errors
    mean_errors = []
    # for different sizes of batches
    for batch in [5, 10, 100, 500]:
        # create plot
        plt.figure()
        # compute parameters and losses with stochastic GD
        w, errors = ridge_regression_SGD(X, y, lmbda=0.1, l_rate=0.1, batch=batch)
        # compute mse
        mean_errors.append(mean_squared_error(y, X.dot(w)))
        # plot losses
        plt.plot(list(range(len(errors))), errors)
        plt.title('SGD Ridge regression loss with batch size {}'.format(batch))
        plt.xlabel('Number of iterations')
        plt.ylabel('Loss')
        plt.show()
    # create plot for mse of ridge with parameters got while training SGD
    plt.figure()
    plt.plot([5, 10, 100, 500], mean_errors)
    plt.title('MSE')
    plt.xlabel('Batch size')
    plt.ylabel('Loss')
    plt.show()

if __name__ == '__main__':
    plt.interactive(False)

    # set seeds to get repeatable results.
    np.random.seed(491)

    # # EXP1: training testing.
    # exp1()
    #
    # # EXP2: generalization performance: increase sample size.
    # exp2()
    #
    # # EXP3: generalization performance: increase dimensionality.
    # exp3()
    #
    # # EXP4: computational complexity by varing dimensions.
    # exp4()
    first_part_assignment()
    second_part_assignment()
```