PRML03

March 4, 2022

1 Report

1.1 Question-1

We have predicted the values from test set datapoints using regularised least square kernel regression model with the RBF kernels. During training process, the kernel size will be 400*401.

We will get coeffiecient vector of size 401*1.

We'll further make test kernel using the combinations of testing data points and training data points of size 200*401.

We'll use this testing kernel and calculated coefficient vector to predict the Y values in test set.

Observations - As we'll increase the value of sigma, the RMSE will also increase but after a certain point it'll start decreasing. Similarily, when we increase the value of lambda, again the RMSE start increasing. But we can not take a very small value of both as well as the tradeoff occurs. (See figures in section Question-1)

1.2 Question-2

We have used the gradient descent method in case of both regularised polynomial regression for M = 1,2 and 5 and regularised kernel regression. I have created a separate function to calculate the gradient. I have initialised the coeffiecients from 1 and keep on updating it by moving in the opposite direction of the gradient.

We further used these coefficients to predict Y values for test data points.

We have plotted the gradient norm vs number of iterations curve for each case as well.

To make the graph more readable, we have left first 1500 values of gradient norms because initially the gradient is decreasing very rapidly in initial few iterations only. thus, graph was becoming as a straight line parallel to Y axis and the a line parallel to x-axis. (see figures in section Question-2 for more details.)

1.3 Question-3

We have used the stochastic gradient descent method in case of both regularised polynomial regression for M = 1,2 and 5 and regularised kernel regression. I have created a separate function to calculate the stochastic gradient. The function is taking 16 datapoints randomly. I have initialised the coeffiecients from 1 and keep on updating it by moving in the opposite direction of the gradient.

We further used these coefficients to predict Y values for test data points.

We have plotted the stochastic gradient norm vs number of iterations curve for each case as well.

To make the graph more readable, we have left first 1500 values of gradient norms because initially the gradient is decreasing very rapidly in initial few iterations only. thus, graph was becoming as a straight line parallel to Y axis and the a line parallel to x-axis. (see figures in section Question-3 for more details.)

1.4 Question-4

We have plotted a bar chart to compare the RMSE values between Normal least square regularised polynomial regression (RPR), regularised least square kernel regression(RKR), RPR and RKR using gradient descent and RPR and RKR using stochastic gradient descent. We have also used 3 different orders M = 1,2 and 5

Eta value used was 0.00002 and tolerance is 2.0. That is why least squared polynomial regression model is giving better result than GD and SGD as the GD and SGD models are not tuned as per their hyper parameter. (see figure in section Question-4 for more details)

1.5 Question-5

We were asked to study the behaviour of eta regarding the convergence. For this I have taken 3 different values of eta = 0.00002, 0.0002 and 0.002 and applied it on regularised polynomial regression with M = 1,2,5 and calculated the RMSE and number of iteration required to converge till the actual solution. In case of smaller step length(eg eta = 0.00002), it takes a long time to converge to the actual solution. As we keep increasing the value of step length, number of iteration required to reach the actual solution decreases.(for eta = 0.0002)

Observations - In case if we further increase the step size value(for eta = 0.002), there is a possibilty that function would miss the minima region and values would overshoot. Similar thing has happened in Question-5 in case of eta=0.002. (check section Question-5 and resultant values for more detail)

2 Creating Datasets

[2]:
```
import numpy as np
import math
```

[3]:
```
##### generating the uniformly distributed X values for bivariate case

def generate_bivariate_X(N):
  X = []
  for i in range(N):
    X = np.random.uniform(size = N)
  return X
```

[4]:
```
##### generating the Y values for bivariate case based on the given X1 and X2␣
 ↪values

def generate_bivariate_Y(X1, X2, N):
  PI = math.pi
  Exp = 2 * PI * ((X1**2 + X2**2) ** 0.5)
  Epsilon = np.random.normal(0, 0.25, size = N)
```

```
  # print(Epsilon)
  Y = np.sin(Exp) + Epsilon
  return Y
```

[5]:
```
##### Making the bivariate dataset for given X1, X2 and Y

def create_XY_bivariate_set(X1, X2, Y):
  result = np.array(list(zip(X1, X2, Y)))
  # print(result)
  return result
```

[6]:
```
x1 = np.array(sorted(generate_bivariate_X(400)))
x2 = np.array(sorted(generate_bivariate_X(400)))
y = generate_bivariate_Y(x1, x2, 400)

training_set_bivariate = create_XY_bivariate_set(x1, x2, y)
training_set_bivariate.shape
```

[6]:
```
(400, 3)
```

[7]:
```
x1 = np.array(sorted(generate_bivariate_X(200)))
x2 = np.array(sorted(generate_bivariate_X(200)))
y = generate_bivariate_Y(x1, x2, 200)

testing_set_bivariate = create_XY_bivariate_set(x1, x2, y)
testing_set_bivariate.shape
```

[7]:
```
(200, 3)
```

[8]:
```
training_set_bivariate[0:5]
```

[8]:
```
array([[ 0.00034608,  0.00037812,  0.08668738],
       [ 0.00159676,  0.00105856,  0.21046033],
       [ 0.00219869,  0.00269573, -0.28029375],
       [ 0.00425768,  0.00422625, -0.19637312],
       [ 0.00593377,  0.0125423 , -0.17338742]])
```

[9]:
```
testing_set_bivariate[0:5]
```

[9]:
```
array([[ 2.91295323e-04,  9.30708094e-04, -1.17278732e-01],
       [ 1.12250314e-02,  7.09644464e-03, -7.01682168e-02],
       [ 2.43722817e-02,  1.16007609e-02, -8.98500029e-02],
       [ 2.63971238e-02,  1.26979676e-02,  5.20807800e-01],
       [ 3.86995438e-02,  1.39721168e-02,  3.21491345e-01]])
```

[10]:
```
len(testing_set_bivariate)
```

[10]:
```
200
```

3 Question-1

Obtain the prediction on testing set and compute the RMSE for regularized least squares kernel regression model with the RBF kernel K(x,y) = exp(- (||x- y|| 2 /). using the direct method. Study the behavior of the kernel parameter and regularization on prediction in terms of RMSE.

[11]:
```
def calculate_kernel_block(X1, Y1, X2, Y2, sigma):
  value = math.exp((-1* ((X1-X2)**2 + (Y1-Y2)**2)) / sigma)
  return value
```

[12]:
```
def construct_kernel(row_X, col_X, sigma):
  kernel = np.zeros((len(row_X), len(col_X)+1))
  kernel[:,-1] = 1
  for i in range(len(row_X)):
    for j in range(len(col_X)):
      kernel[i][j] = calculate_kernel_block(row_X[i][0], row_X[i][1],␣
 ↪col_X[j][0], col_X[j][1], sigma)
  return kernel
```

[13]:
```
def calculate_U(kernel, Y, lamb):
  result = np.linalg.inv((kernel.T @ kernel) + lamb * np.
 ↪identity(int(len(kernel[0])))) @ kernel.T @ Y
  return result
```

[14]:
```
###### Function to calculate the RMSE based on actual and estimated Y values

def calculate_RMSE(Y, Y_real):
  error = np.subtract(Y, Y_real)
  squared_error = np.square(error)
  total_squared_error = squared_error.sum()     #### to sum all the values in␣
 ↪each row
  total_mean_squared_error = total_squared_error/len(Y)
  total_root_mean_squared_error = np.sqrt(total_mean_squared_error)
  return total_mean_squared_error
```

[15]:
```
train_X = training_set_bivariate[:, 0:2]
train_Y = training_set_bivariate[:, -1]
test_X = testing_set_bivariate[:, 0:2]
test_Y = testing_set_bivariate[:, -1]
```

[16]:
```
sigma = 5

train_kernel = construct_kernel(train_X, train_X, sigma)
```

[17]:
```
train_kernel.shape
```

[17]:
```
(400, 401)
```

[18]:
```
lamb = 0.001

U = calculate_U(train_kernel, train_Y, lamb)
```

[19]:
```
U.shape
```

[19]:
```
(401,)
```

[20]:
```
train_Y_est = train_kernel @ U
train_RMSE = calculate_RMSE(train_Y_est, train_Y)
train_RMSE
```

[20]:
```
0.2857097461006154
```

[21]:
```
test_kernel = construct_kernel(test_X, train_X, sigma)
```

[22]:
```
test_kernel.shape
```

[22]:
```
(200, 401)
```

[23]:
```
test_Y_est = test_kernel @ U
```

[24]:
```
test_Y_est.shape
```

[24]:
```
(200,)
```

[25]:
```
print(test_Y_est)
```

```
[ 1.10236187e+00  1.05739684e+00  1.00001966e+00  9.91460192e-01
  9.33769451e-01  9.24585204e-01  8.24499338e-01  8.17247932e-01
  7.84489253e-01  7.73783939e-01  7.22867261e-01  7.07957359e-01
  7.17630143e-01  6.80801517e-01  6.83699890e-01  6.80245961e-01
  6.56662580e-01  6.09374181e-01  5.91335081e-01  5.90338480e-01
  5.46449684e-01  5.39337327e-01  4.95102229e-01  4.94878982e-01
  4.92545585e-01  4.68582628e-01  4.68388777e-01  4.63800480e-01
  3.99685402e-01  3.91917473e-01  3.89003925e-01  3.00504541e-01
  2.82220532e-01  2.90291073e-01  2.68881085e-01  2.45852362e-01
  2.22253738e-01  1.77057749e-01  1.51296927e-01  1.53406724e-01
  1.14554736e-01  8.88692712e-02  7.33289860e-02  7.33332911e-02
  7.31983487e-02  2.15910316e-02 -6.51734760e-03 -2.23396075e-02
 -2.40680080e-02  2.25146425e-03 -1.99541219e-02 -4.45257749e-02
 -1.78652917e-02  2.49139371e-02  3.21297710e-02 -5.15141017e-03
 -6.87817361e-03 -1.07827845e-02 -1.75684170e-02  1.70183483e-03
 -4.96931998e-02 -5.02243410e-02 -6.41213253e-02 -6.23334360e-02
 -9.94274072e-02 -9.10131492e-02 -1.19710760e-01 -1.33177294e-01
 -1.28851469e-01 -1.28007984e-01 -1.60521672e-01 -1.68337265e-01
 -1.56014637e-01 -1.41162258e-01 -1.25712673e-01 -1.35645211e-01
 -1.29991352e-01 -1.18894035e-01 -1.18355057e-01 -1.07619947e-01
 -1.37140082e-01 -2.29596181e-01 -2.39306771e-01 -2.87264263e-01
 -2.90744470e-01 -2.98464766e-01 -2.88402910e-01 -2.79480014e-01
 -2.84713034e-01 -2.81116953e-01 -2.50431964e-01 -2.54706300e-01
 -2.38903131e-01 -2.12681843e-01 -2.06190190e-01 -2.03538932e-01
 -1.60333415e-01 -1.62225678e-01 -1.57175717e-01 -1.67397676e-01
 -1.56207465e-01 -1.51186143e-01 -1.45746792e-01 -1.29891250e-01
 -9.69981698e-02 -1.22205610e-01 -1.14746958e-01 -1.08292716e-01
 -1.11351394e-01 -1.02833203e-01 -1.04309343e-01 -1.19233463e-01
```

```
-9.95704173e-02 -9.61291021e-02 -9.57158382e-02 -8.06914797e-02
-6.76460273e-02 -6.75536920e-02 -7.69812396e-02 -6.21548584e-02
-5.79155704e-02 -5.41030760e-02 -4.77028386e-02 -7.02497270e-02
-7.35401639e-02 -9.09890968e-02 -6.48331856e-02 -5.96186525e-02
-6.61268828e-02 -6.36373416e-02 -3.08931522e-02 -4.74845469e-02
-3.51588098e-02 -2.17761279e-03  6.36433539e-03  8.89457152e-03
 4.69220091e-03  9.91177258e-03  2.42675038e-04  1.26324539e-02
 1.21378792e-02  1.36378280e-02  4.83301845e-02  8.67811732e-02
 9.84075184e-02  1.00867458e-01  9.16925749e-02  1.06799445e-01
 1.54787248e-01  2.47345137e-01  2.53849209e-01  2.45731053e-01
 2.51311923e-01  2.57891653e-01  2.56563566e-01  2.61346779e-01
 2.60028365e-01  2.76730648e-01  2.93011040e-01  3.17181264e-01
 3.05717829e-01  3.76494336e-01  3.72414594e-01  3.76082636e-01
 4.26325790e-01  4.32926344e-01  4.96497181e-01  4.96319601e-01
 5.15345514e-01  5.44521692e-01  5.87848776e-01  6.11112078e-01
 6.15029715e-01  6.17969171e-01  6.37675786e-01  6.53730514e-01
 6.58267139e-01  6.59179402e-01  7.54466434e-01  7.66038265e-01
 7.64294452e-01  7.64149926e-01  8.21840412e-01  8.46570504e-01
 8.98414164e-01  9.22513272e-01  9.22642433e-01  9.26876845e-01
 9.44606966e-01  9.66852895e-01  9.75719821e-01  1.01490860e+00
 1.09953995e+00  1.09552220e+00  1.09879164e+00  1.14268800e+00
 1.23895942e+00  1.24169840e+00  1.24354179e+00  1.24323778e+00]
```

[26]:
```
test_RMSE = calculate_RMSE(test_Y_est, test_Y)
```

[27]:
```
test_RMSE_RKR_Q1 = test_RMSE
test_RMSE_RKR_Q1
```

[27]:
```
0.41077051829435407
```

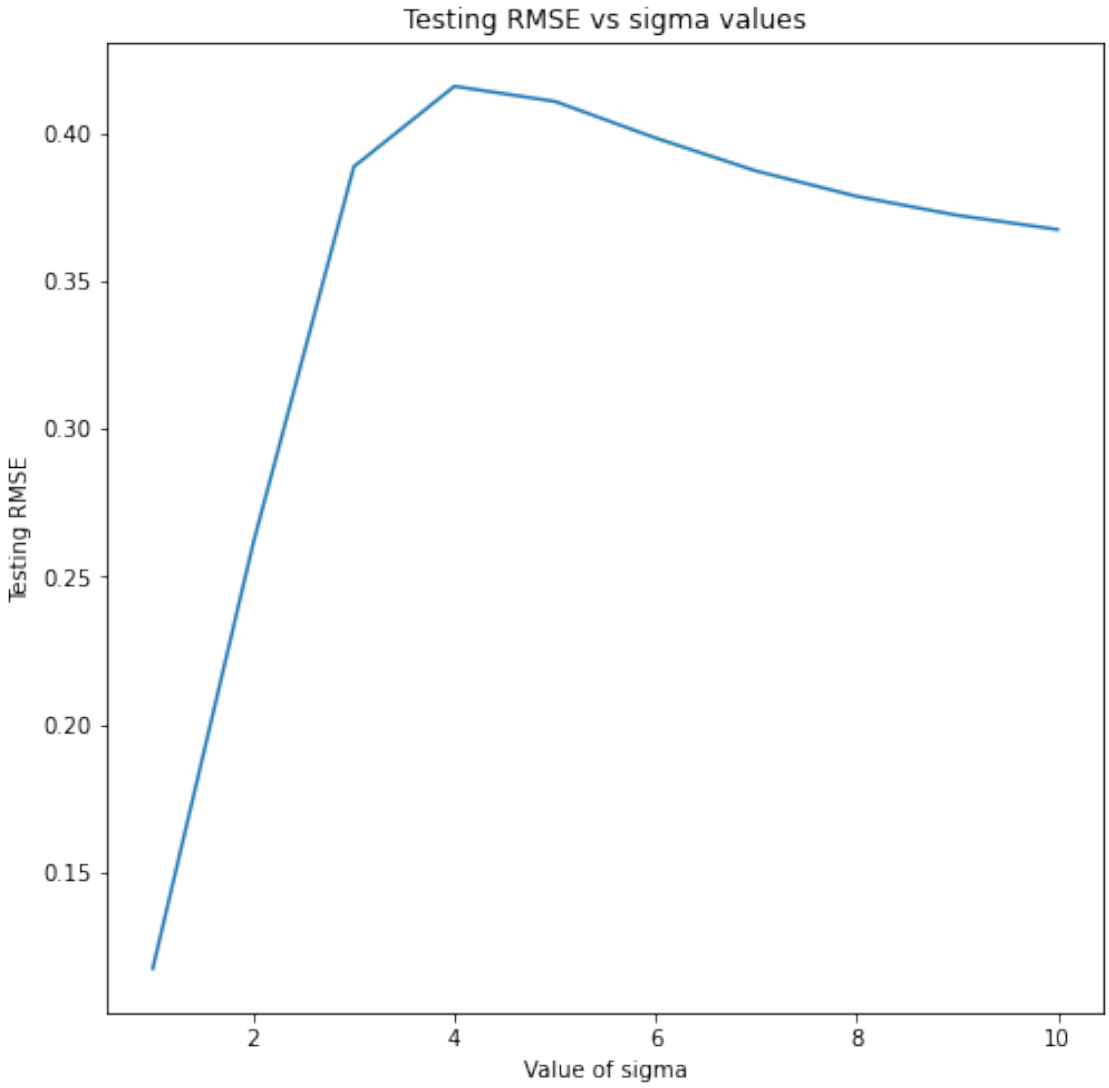
3.0.1 Behaviour of sigma and lambda on predictions in terms of RMSE

[28]:
```
sigma_list = list(range(1,11,1))
lambda_list = [0.1, 0.05, 0.01, 0.005, 0.001]

print('comparison of Behaviour of sigma and lambda on predictions in terms of␣
 ↪RMSE=======')

RMSE_sigma = []
RMSE_lambda = []

for sigma in sigma_list:
  lamb = 0.001
  train_kernel = construct_kernel(train_X, train_X, sigma)
  test_kernel = construct_kernel(test_X, train_X, sigma)

  U = calculate_U(train_kernel, train_Y, lamb)
  train_Y_est = train_kernel @ U
```

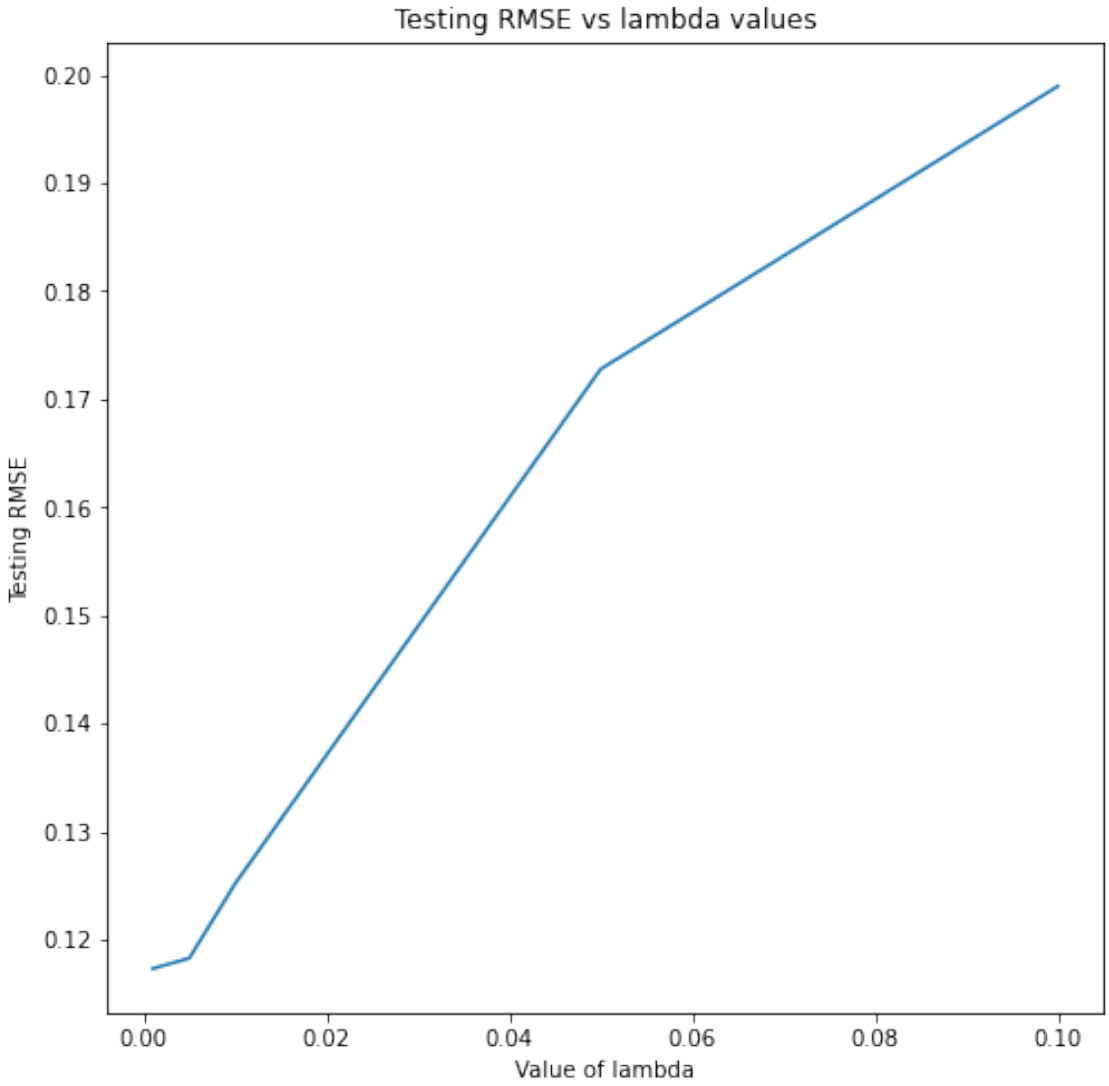
```
  train_RMSE = calculate_RMSE(train_Y_est, train_Y)
  test_Y_est = test_kernel @ U
  test_RMSE = calculate_RMSE(test_Y_est, test_Y)
  RMSE_sigma.append(test_RMSE)
  # print()
  # print('Train RMSE value in case of sigma value {} and lambda value {} is :␣
 ↪{}'.format(sigma, lamb, train_RMSE))
  # print('Test RMSE value in case of sigma value {} and lambda value {} is :␣
 ↪{}'.format(sigma, lamb, test_RMSE))
  #␣
 ↪print('================================================================================

for lamb in lambda_list:
  sigma = 1
  train_kernel = construct_kernel(train_X, train_X, sigma)
  test_kernel = construct_kernel(test_X, train_X, sigma)

  U = calculate_U(train_kernel, train_Y, lamb)
  train_Y_est = train_kernel @ U
  train_RMSE = calculate_RMSE(train_Y_est, train_Y)
  test_Y_est = test_kernel @ U
  test_RMSE = calculate_RMSE(test_Y_est, test_Y)
  RMSE_lambda.append(test_RMSE)
```

comparison of Behaviour of sigma and lambda on predictions in terms of RMSE=======

[29]:
```
import matplotlib.pyplot as plt

fig1 = plt.figure(figsize = (8,8))
plt.plot(sigma_list, RMSE_sigma)
plt.xlabel('Value of sigma')
plt.ylabel('Testing RMSE')
plt.title('Testing RMSE vs sigma values')
plt.show()
plt.close(fig1)
```



[30]:
```
fig1 = plt.figure(figsize = (8,8))
plt.plot(lambda_list, RMSE_lambda)
plt.xlabel('Value of lambda')
plt.ylabel('Testing RMSE')
plt.title('Testing RMSE vs lambda values')
plt.show()
plt.close(fig1)
```



4 Question-2

Obtain the prediction on testing set and compute the RMSE for regularized polynomial regression models for order M =1,2 , 5 and regularized least squares kernel regression model using the gradient descent method.

4.0.1 first implementing Regularised polynomial regression model for order M=1,2,5 to calculate RMSE on testing data-points using gradient-descent method

[31]:
```
train_X1, train_X2 = zip(*train_X)
train_X1 = np.array(train_X1)
train_X2 = np.array(train_X2)
```

[32]:
```
test_X1, test_X2 = zip(*test_X)
test_X1 = np.array(test_X1)
test_X2 = np.array(test_X2)
```

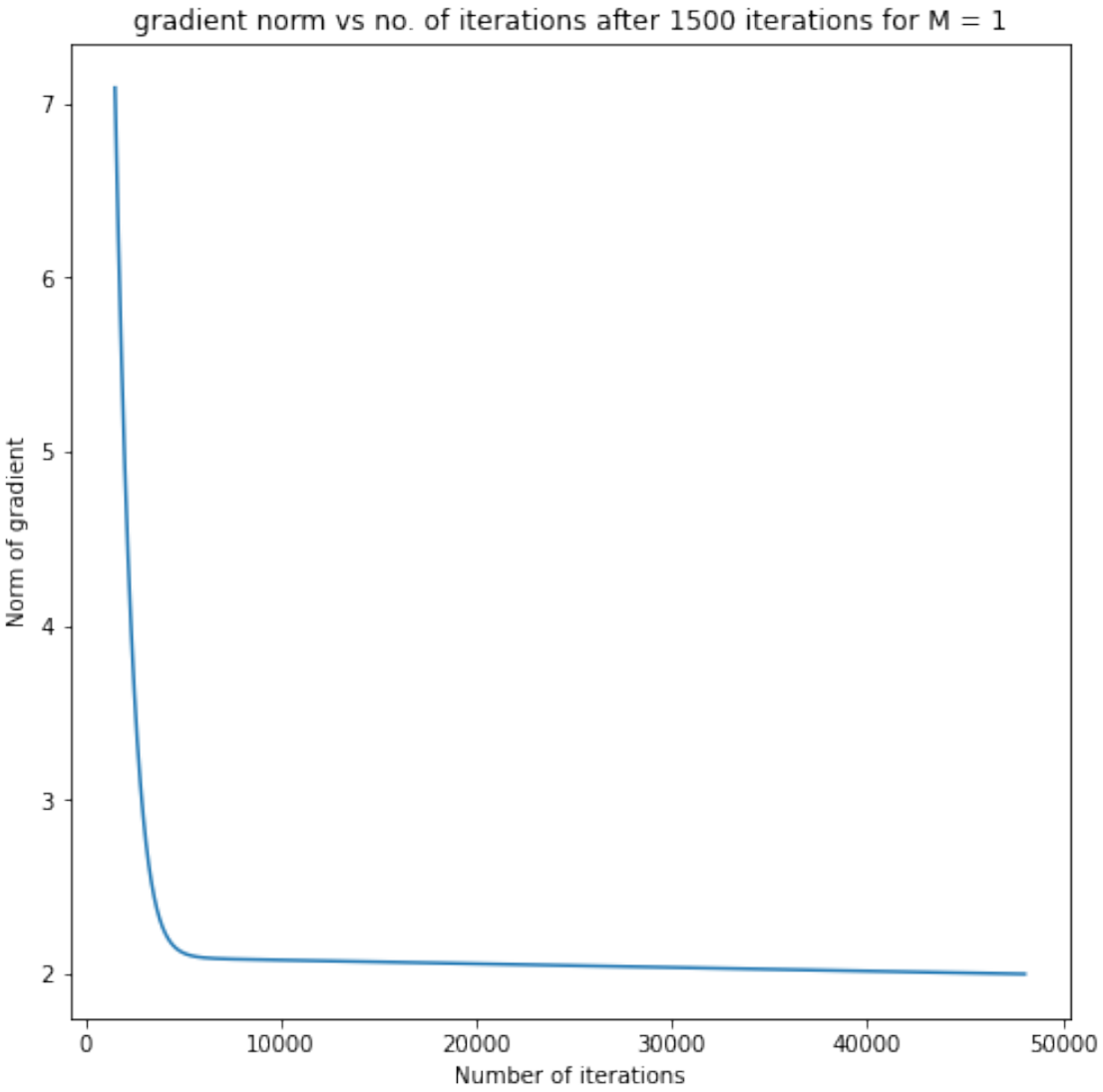
[33]:
```
def compute_gradient(lamb, U, A, Y):
  grad = (lamb*U) - (A.T @ (Y - (A@U)))
  return grad
```

[34]:
```
def generate_single_col_vector(x, y, M):
  v = []
  for j in range(M+1):
    x_pow = j
    y_pow = 0
    while x_pow >= 0:
      v.append((x**x_pow) * (y**y_pow))
      x_pow = x_pow-1
      y_pow = y_pow+1
  return v
```

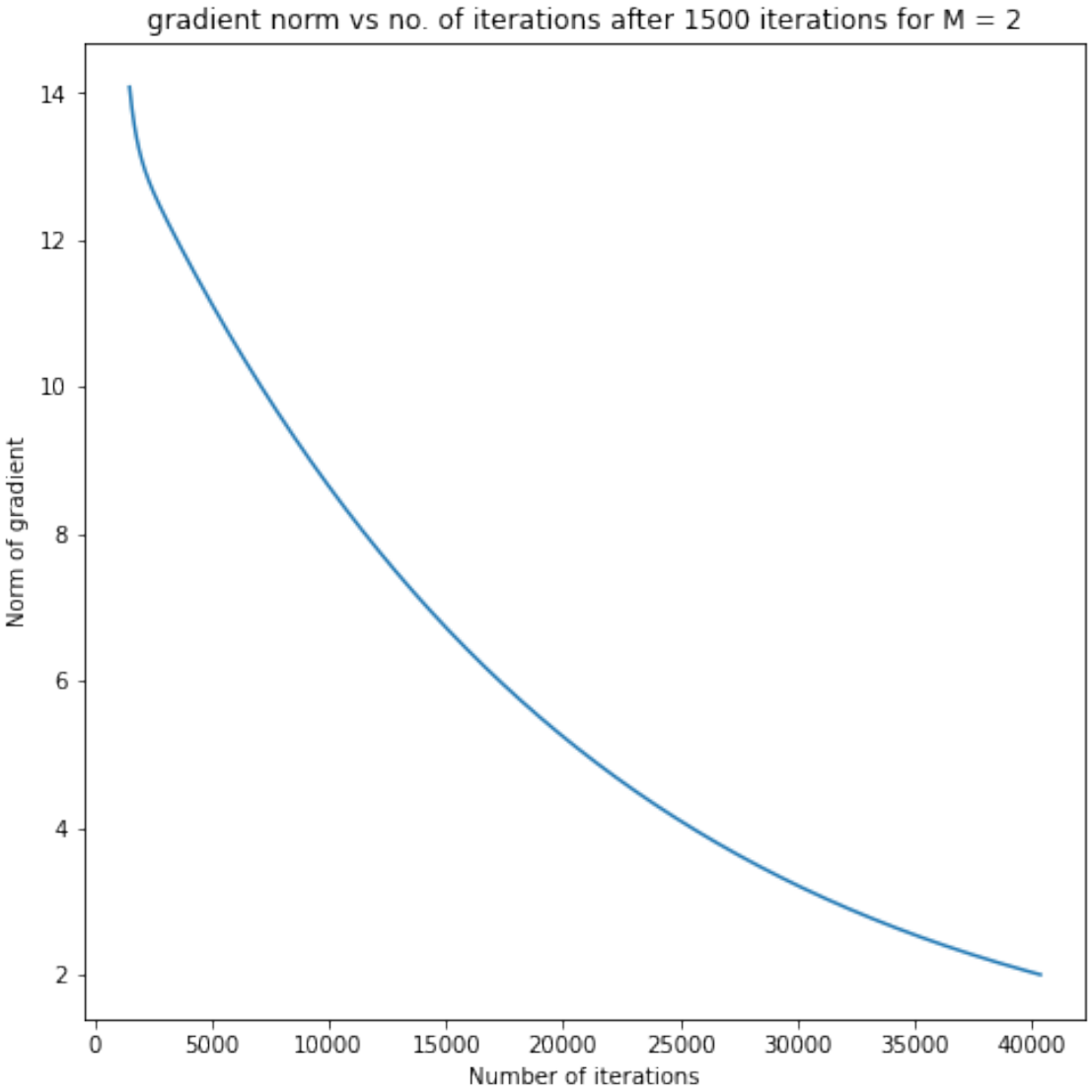
[35]:
```
def generate_bivariate_x_matrix(X1, X2, M):
  rows = int((M+1)*(M+2)/2)
  x_matrix = np.zeros((len(X1), rows))
  for j in range(len(X1)):
    single_col_vector = generate_single_col_vector(X1[j], X2[j], M)
    x_matrix[j,:] = single_col_vector
  return x_matrix
```

[70]:
```
M_list = np.array([1,2,5])
values_in_eq = (M_list+1) * (M_list+2)/2
RMSE_RPR = []

lamb = math.exp(-18)
eta = 0.00002
tolerance = 2.0
### 0.05

for i in range(len(values_in_eq)):
  M = M_list[i]
  val = values_in_eq[i]

  x_matrix_bivariate = generate_bivariate_x_matrix(train_X1, train_X2, M)
  x_matrix_bivariate_test = generate_bivariate_x_matrix(test_X1, test_X2, M)
  weight_coeff = np.array([1]*len(x_matrix_bivariate[0]))

  counter = 1
  G_norm = []

  G = compute_gradient(lamb, weight_coeff, x_matrix_bivariate, train_Y)
```

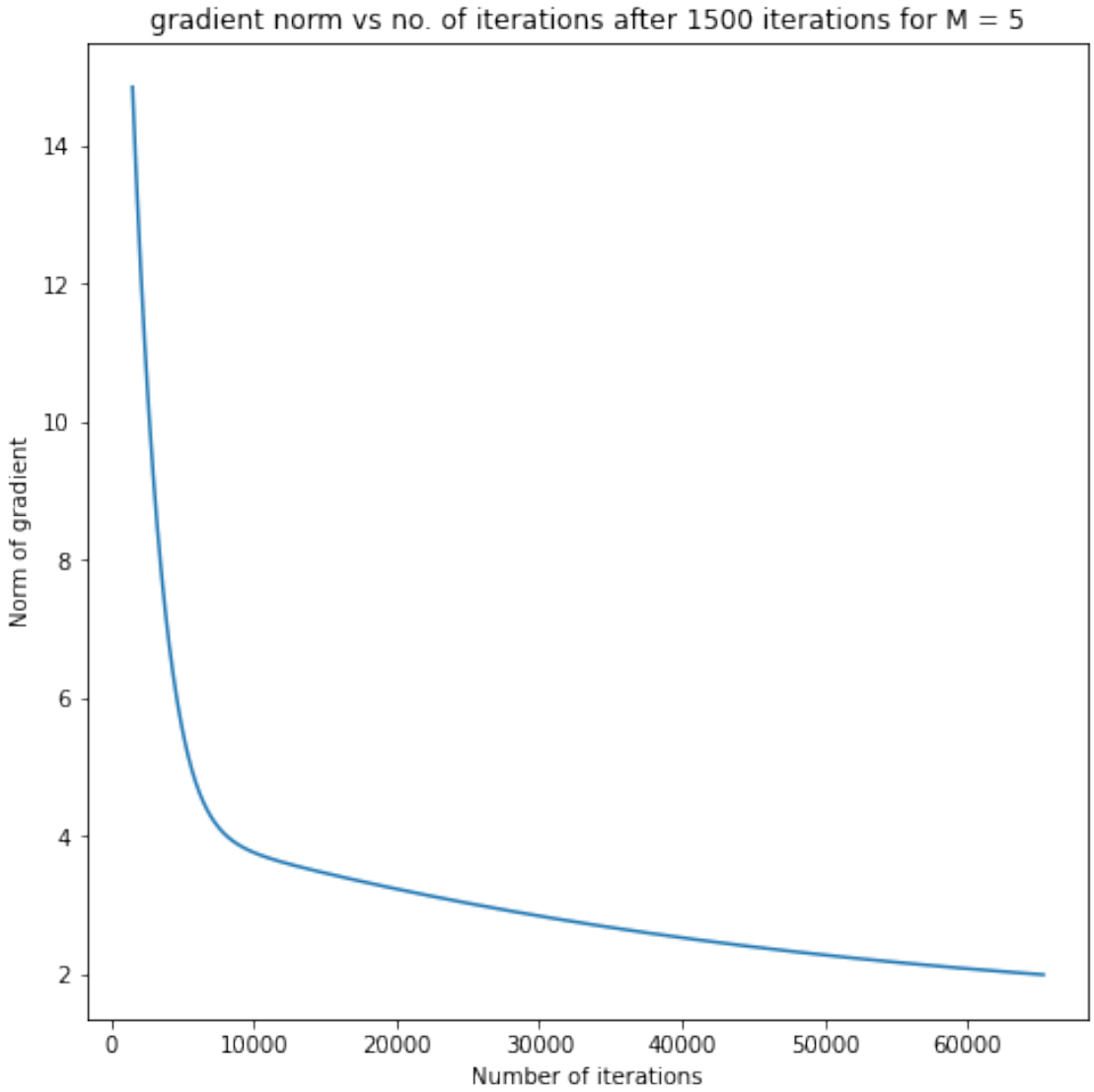
```
 while np.linalg.norm(G) > tolerance:
   weight_coeff = weight_coeff - eta * G
   G_norm.append(np.linalg.norm(G))
   G = compute_gradient(lamb, weight_coeff, x_matrix_bivariate, train_Y)

   # print('Iteration no : '+str(counter))
   counter = counter+1
   # print('Current gradient\'s norm value is : '+str(np.linalg.norm(G)))
   # print('================================================================')

 test_Y_est_RPR = x_matrix_bivariate_test @ weight_coeff
 RMSE_current_M = calculate_RMSE(test_Y_est_RPR, test_Y)

 fig2 = plt.figure(figsize = (8,8))
 x_axis = list(range(counter-1))
 plt.plot(x_axis[1500:], G_norm[1500:])
 plt.xlabel('Number of iterations')
 plt.ylabel('Norm of gradient')
 plt.title('gradient norm vs no. of iterations after 1500 iterations for M =␣
↪'+str(M))
 plt.show()
 plt.close(fig2)

 print('Testing RMSE for M={} is : {}'.format(M, RMSE_current_M))
 print('Total number of iterations performed to get the values less then the␣
↪tolerance are : '+str(counter-1))
␣
↪print('================================================================================')

 RMSE_RPR.append(RMSE_current_M)
```




```
Testing RMSE for M=1 is : 0.6988961589749403
Total number of iterations performed to get the values less then the tolerance
are : 48055
================================================================================
```




```
Testing RMSE for M=2 is : 0.3949235200352556
Total number of iterations performed to get the values less then the tolerance
are : 40376
================================================================================
```




```
Testing RMSE for M=5 is : 0.3362408908496466
Total number of iterations performed to get the values less then the tolerance
are : 65320
================================================================================
```

[37]:
```
RMSE_RPR
```

[37]:
```
[0.6988961589749403, 0.3949235200352556, 0.3362408908496466]
```

4.0.2 Now implementing Regularised least square Kernel regression model using gradient descent to calculate RMSE on testing data-points

[38]:
```
##### I am using the functions defined in question-1 to build the training and␣
 ↪testing kernels

sigma = 1

train_kernel = construct_kernel(train_X, train_X, sigma)
test_kernel = construct_kernel(test_X, train_X, sigma)
```

[ ]:
```
coeff = np.array([1]*len(train_kernel[0]))

lamb = math.exp(-18)
eta = 0.00002
tolerance = 2.0
### 0.05
counter = 1
G_norm = []
G = compute_gradient(lamb, coeff, train_kernel, train_Y)

while np.linalg.norm(G) > tolerance:
  coeff = coeff - eta * G
  G_norm.append(np.linalg.norm(G))
  G = compute_gradient(lamb, coeff, train_kernel, train_Y)

  print('Iteration no : '+str(counter))
  counter = counter+1
  print(G.shape)
  print('Current gradient\'s norm value is : '+str(np.linalg.norm(G)))
  print('==============================================================')
```

[40]:
```
test_Y_est_Q2 = test_kernel @ coeff
test_RMSE_RKR_Q2 = calculate_RMSE(test_Y_est_Q2, test_Y)
print('Testing RMSE in case of regularized kernel regression is : {}'.
 ↪format(test_RMSE_RKR_Q2))
print('Total number of iterations performed to get the values less then the␣
 ↪tolerance are : '+str(counter-1))
print('================================================================================')
```

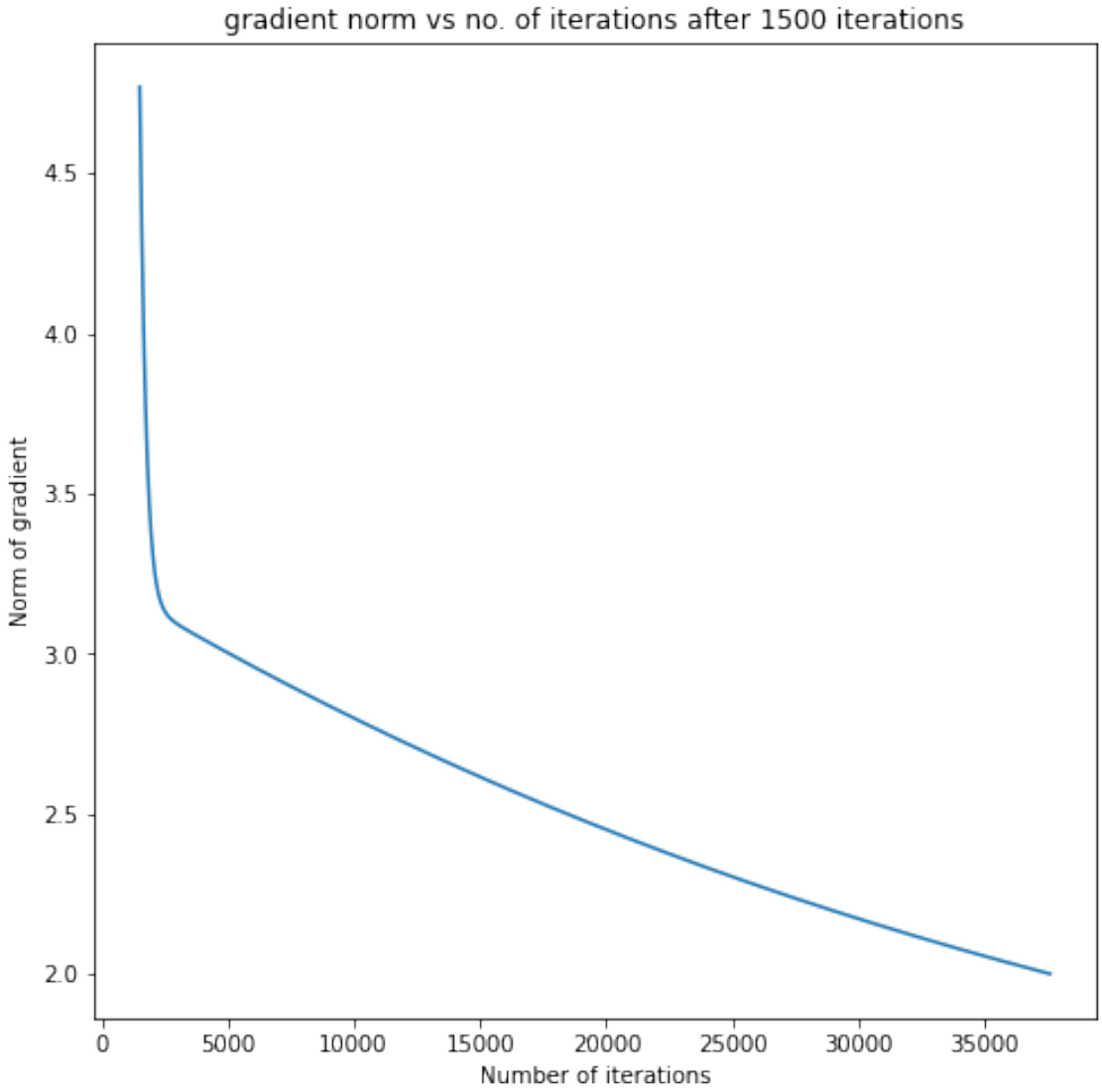
```
Testing RMSE in case of regularized kernel regression is : 0.24981980241387333
Total number of iterations performed to get the values less then the tolerance
are : 37595
================================================================================
```

[41]:
```
print(test_Y_est_Q2)
```

```
[ 0.98886352  0.97415667  0.95735372  0.95430514  0.93968982  0.92905828
  0.88812318  0.88572217  0.87585754  0.87181263  0.85403544  0.83888763
```

```
 0.82456049  0.8125957   0.80874911  0.80387845  0.7959027   0.77425223
 0.76133723  0.76019525  0.7302087   0.72536218  0.70504479  0.70340622
 0.67607002  0.66014159  0.65801794  0.6548083   0.63134912  0.62476498
 0.62118572  0.58703334  0.57286485  0.54093394  0.51679997  0.50033684
 0.49064058  0.46506181  0.45234553  0.4388773   0.4207154   0.40767846
 0.40021627  0.3939544   0.39255072  0.35484706  0.33972076  0.33024332
 0.32789781  0.29536594  0.27596814  0.26501957  0.226382    0.18795678
 0.18127655  0.16416668  0.15334597  0.13747023  0.12231853  0.10319754
 0.07668341  0.0522996   0.04588322  0.04205093  0.02398863  0.00960589
-0.02697906 -0.03526087 -0.04421372 -0.06493985 -0.08650776 -0.09523434
-0.10822442 -0.12912357 -0.15999602 -0.17026613 -0.17472144 -0.18355527
-0.18857654 -0.20932671 -0.22786684 -0.27295659 -0.28203639 -0.3005455
-0.31037176 -0.31660526 -0.32932418 -0.3342994  -0.34254585 -0.34859346
-0.35923258 -0.36869475 -0.37972607 -0.39048198 -0.39230785 -0.39438526
-0.41009337 -0.41132105 -0.41287555 -0.41639766 -0.41738389 -0.41827029
-0.4199241  -0.42109153 -0.42366699 -0.42975581 -0.43068733 -0.4305144
-0.43082987 -0.43052142 -0.43062577 -0.43145936 -0.42404365 -0.4204254
-0.42028743 -0.41737393 -0.41471534 -0.4139917  -0.41274939 -0.4074769
-0.40605134 -0.40370337 -0.40006063 -0.39202163 -0.39072136 -0.38013096
-0.37131982 -0.36642515 -0.36385945 -0.36271888 -0.35195704 -0.33677999
-0.33245076 -0.30178142 -0.29673993 -0.29508608 -0.28128349 -0.27274308
-0.26039498 -0.2474647  -0.24496135 -0.24135285 -0.21288414 -0.1879794
-0.18239187 -0.17277583 -0.12115479 -0.10394381 -0.07645071 -0.02165812
-0.01700639  0.03983103  0.04749761  0.0657839   0.08290003  0.10660079
 0.12582149  0.15079987  0.17129289  0.18762242  0.21970067  0.26956821
 0.29241645  0.31283713  0.35934093  0.39209999  0.42883833  0.43229564
 0.44497149  0.48211539  0.56380936  0.60765259  0.61625341  0.62617662
 0.64250702  0.66261437  0.66624472  0.67572303  0.73480553  0.74248371
 0.80154418  0.80812808  0.84922034  0.86423709  0.93364087  0.96971189
 0.97580003  0.98304001  1.02000286  1.0336418   1.04189161  1.07587826
 1.1389592   1.17087637  1.17857422  1.20939912  1.26365838  1.27642689
 1.3118159   1.32904625]
```

[42]:
```
###### plotting the gradient norm vs number of itertaions curve

fig1 = plt.figure(figsize = (8,8))
x_axis = list(range(counter-1))
plt.plot(x_axis[1500:], G_norm[1500:])
plt.xlabel('Number of iterations')
plt.ylabel('Norm of gradient')
plt.title('gradient norm vs no. of iterations after 1500 iterations')
plt.show()
```



[43]:
```
plt.close(fig1)
```

5 Question-3

Obtain the prediction on testing set and compute the RMSE for regularized polynomial regression models for order M =1,2 , 5 and regularized least squares kernel regression model using the k-mini batch stochastic gradient descent method.

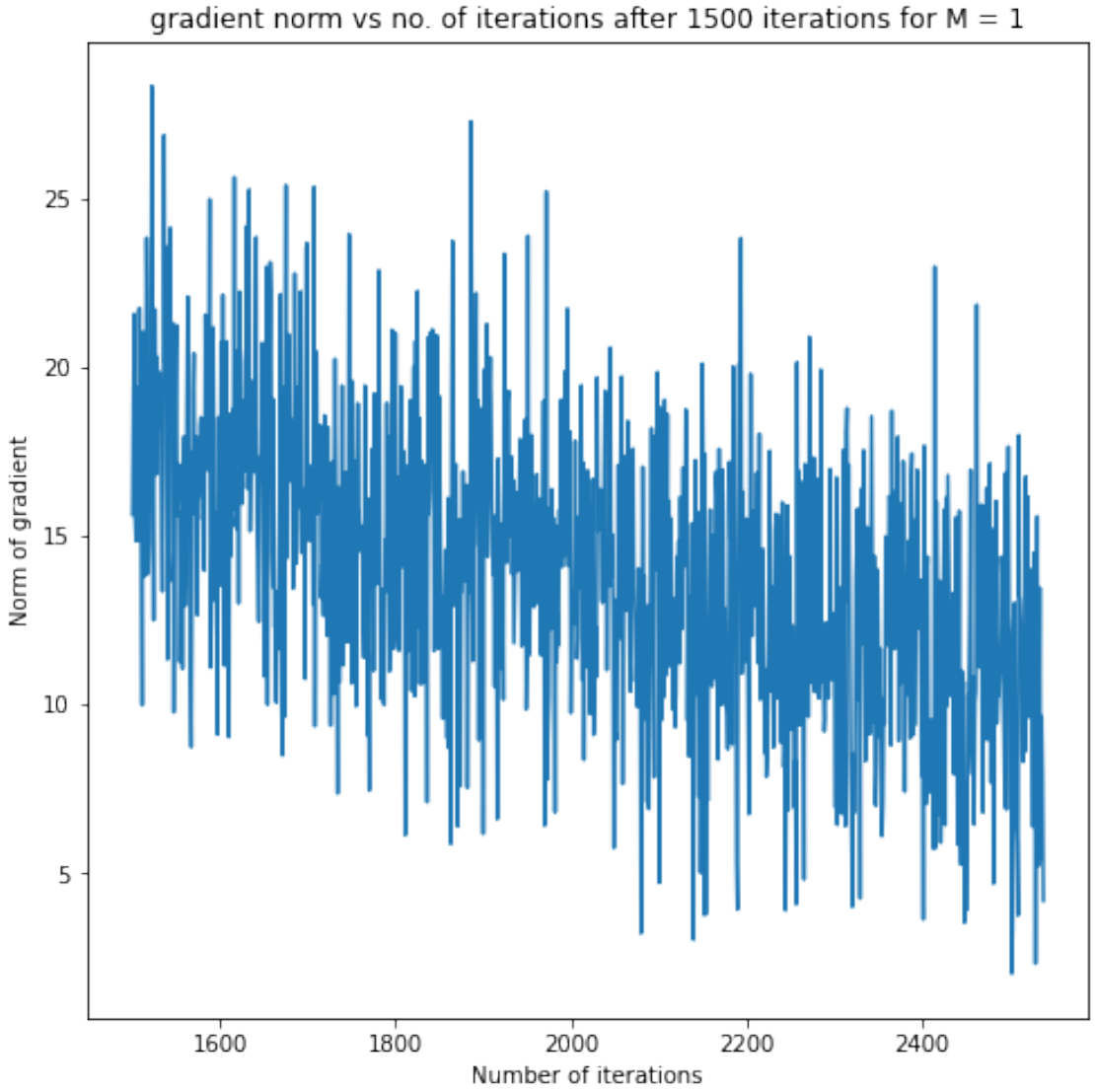
[44]:
```
import random
```

[45]:
```
def compute_gradient_K_batch(lamb, U, A, Y):
  P = random.sample(range(0, 400), 16)
  grad = (lamb*U) - (A[P].T @ (Y[P] - (A[P] @ U)))
```

```
  return grad
```

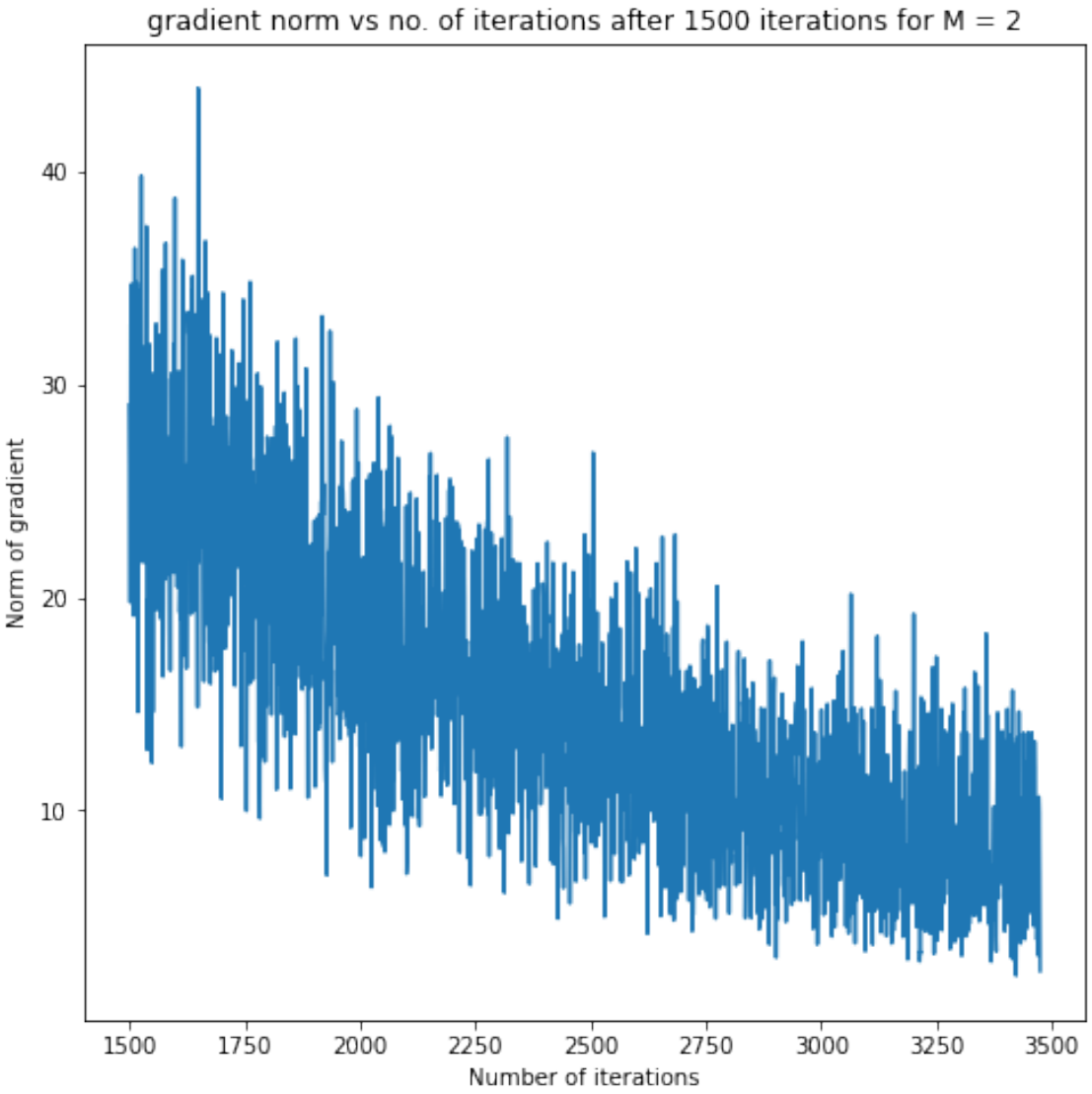
[46]:
```
M_list = np.array([1,2,5])
values_in_eq = (M_list+1) * (M_list+2)/2
RMSE_RPR_K_batch = []

lamb = math.exp(-18)
eta = 0.00002
tolerance = 2.0
### 0.05


for i in range(len(values_in_eq)):
  M = M_list[i]
  val = values_in_eq[i]

  x_matrix_bivariate = generate_bivariate_x_matrix(train_X1, train_X2, M)
  x_matrix_bivariate_test = generate_bivariate_x_matrix(test_X1, test_X2, M)
  weight_coeff = np.array([1]*len(x_matrix_bivariate[0]))

  counter = 1
  G_norm = []

  G = compute_gradient_K_batch(lamb, weight_coeff, x_matrix_bivariate, train_Y)

  while np.linalg.norm(G) > tolerance:
    weight_coeff = weight_coeff - eta * G
    G_norm.append(np.linalg.norm(G))
    G = compute_gradient_K_batch(lamb, weight_coeff, x_matrix_bivariate,␣
↪train_Y)

    # print('Iteration no : '+str(counter))
    counter = counter+1
    # print('Current gradient\'s norm value is : '+str(np.linalg.norm(G)))
    # print('================================================================')

  test_Y_est_RPR_K_batch = x_matrix_bivariate_test @ weight_coeff
  RMSE_current_M = calculate_RMSE(test_Y_est_RPR_K_batch, test_Y)

  fig1 = plt.figure(figsize = (8,8))
  x_axis = list(range(counter-1))
  plt.plot(x_axis[1500:], G_norm[1500:])
  plt.xlabel('Number of iterations')
  plt.ylabel('Norm of gradient')
  plt.title('gradient norm vs no. of iterations after 1500 iterations for M =␣
↪'+str(M))
  plt.show()
```

```
 plt.close(fig1)

 print('Testing RMSE for M={} is : {}'.format(M, RMSE_current_M))
 print('Total number of iterations performed to get the values less then the␣
↪tolerance are : '+str(counter-1))
␣
↪print('================================================================================')

 RMSE_RPR_K_batch.append(RMSE_current_M)
```

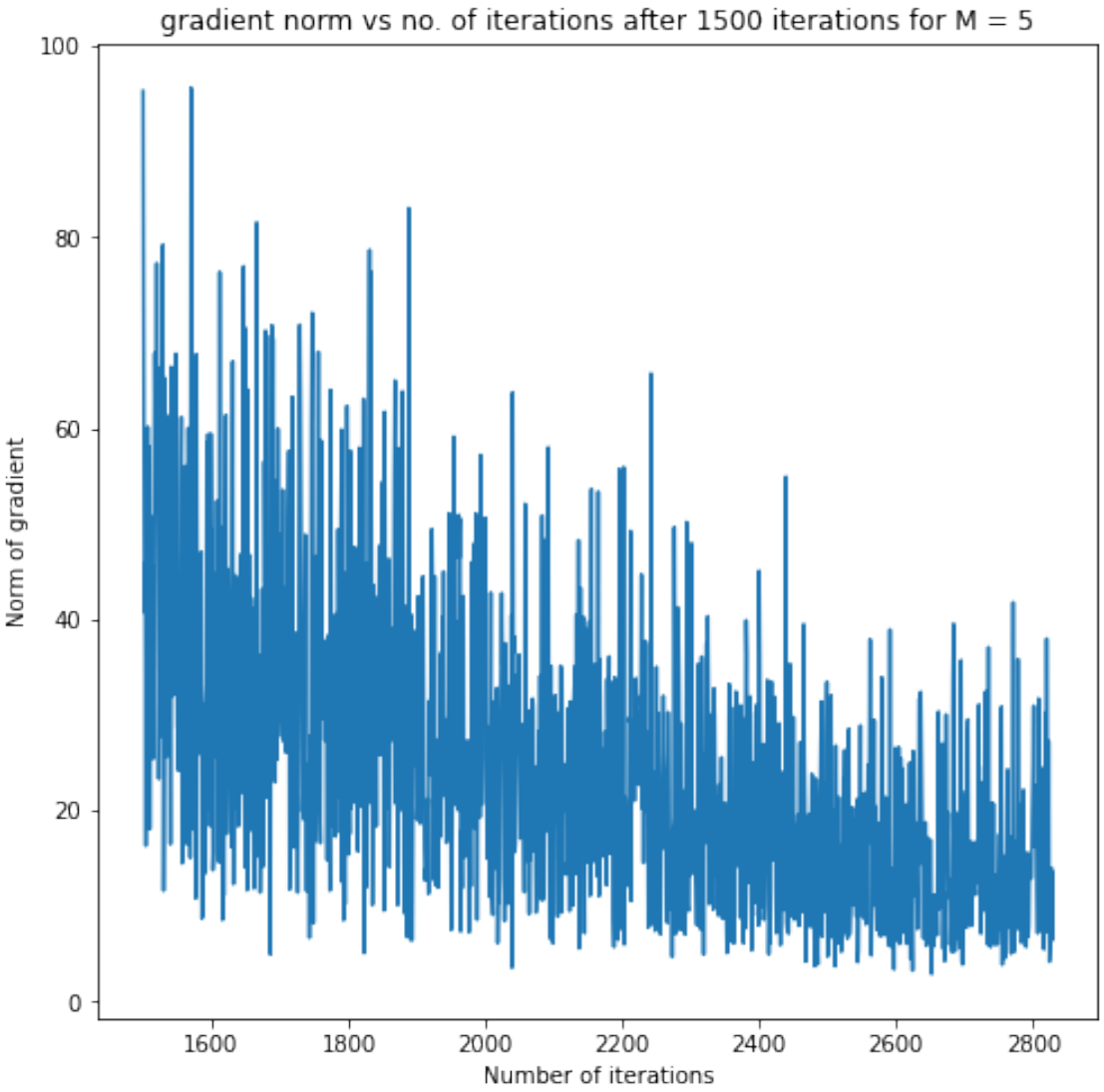



```
Testing RMSE for M=1 is : 0.9719535459359031
Total number of iterations performed to get the values less then the tolerance
```

```
are : 2540
==========================================================================
```




```
Testing RMSE for M=2 is : 0.8029138557465921
Total number of iterations performed to get the values less then the tolerance
are : 3475
==========================================================================
```




```
Testing RMSE for M=5 is : 0.8934166277453599
Total number of iterations performed to get the values less then the tolerance
are : 2830
================================================================================
```

[ ]:
```
coeff = np.array([1]*len(train_kernel[0]))

lamb = math.exp(-18)
eta = 0.00002
tolerance = 2.0
### 0.05
counter = 1
G_norm = []
```

```
G = compute_gradient_K_batch(lamb, coeff, train_kernel, train_Y)

while np.linalg.norm(G) > tolerance:
  coeff = coeff - eta * G
  G_norm.append(np.linalg.norm(G))
  G = compute_gradient_K_batch(lamb, coeff, train_kernel, train_Y)

  print('Iteration no : '+str(counter))
  counter = counter+1
  print(G.shape)
  print('Current gradient\'s norm value is : '+str(np.linalg.norm(G)))
  print('================================================================')
```

[48]:
```
test_Y_est_Q3 = test_kernel @ coeff
test_RMSE_RKR_Q3 = calculate_RMSE(test_Y_est_Q3, test_Y)
print('Testing RMSE in case of regularized kernel regression is : {}'.
 ↪format(test_RMSE_RKR_Q3))
print('Total number of iterations performed to get the values less then the␣
 ↪tolerance are : '+str(counter-1))
print('================================================================================')
```

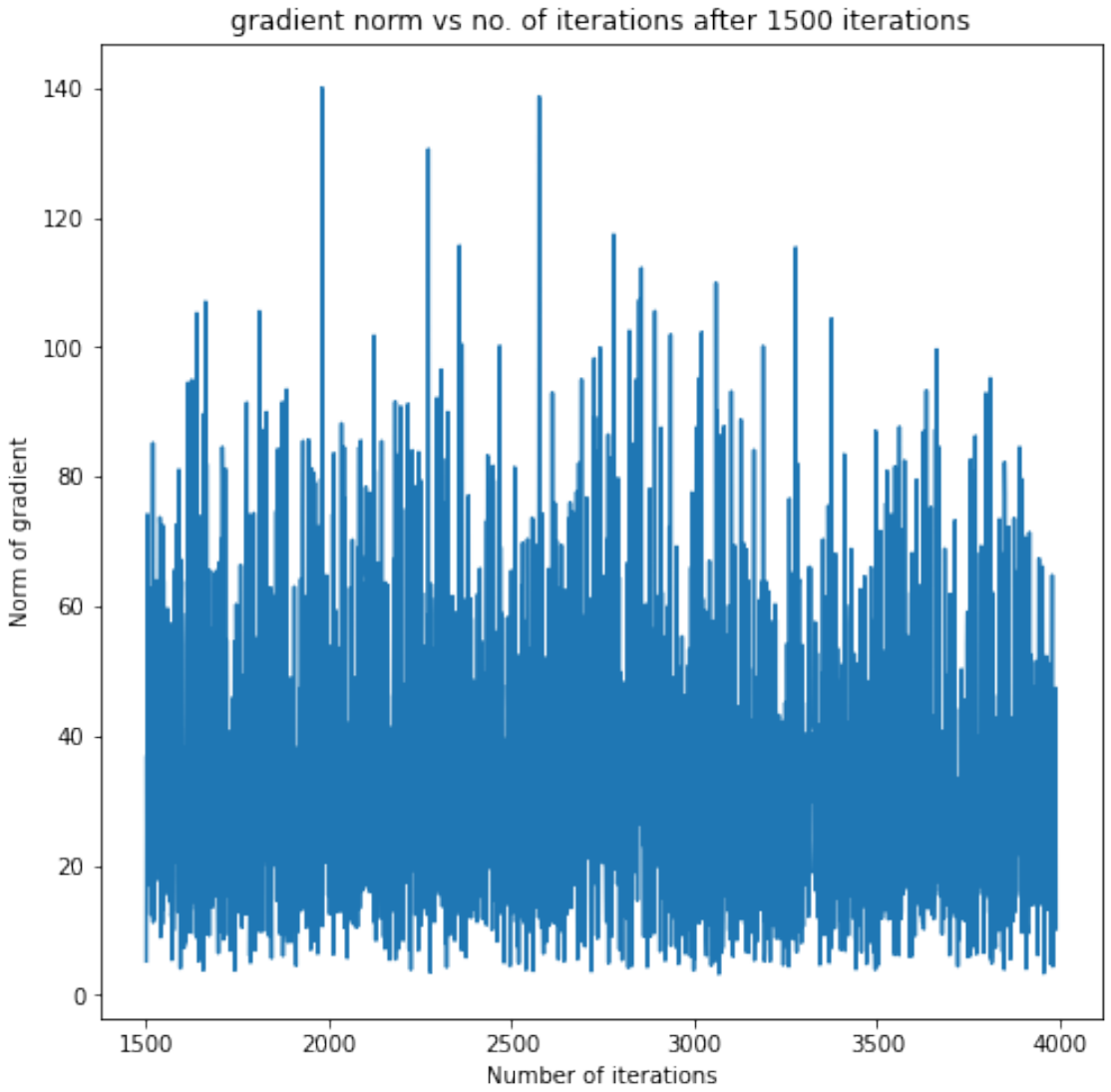
```
Testing RMSE in case of regularized kernel regression is : 0.35803339077543156
Total number of iterations performed to get the values less then the tolerance
are : 3991
================================================================================
```

[49]:
```
print(test_Y_est_Q3)
```

```
[ 1.98913325e+00  1.93145328e+00  1.86952774e+00  1.85860549e+00
  1.80846458e+00  1.77156771e+00  1.64083770e+00  1.63358606e+00
  1.60435553e+00  1.59236327e+00  1.54095371e+00  1.49627778e+00
  1.45370038e+00  1.42126228e+00  1.41015390e+00  1.39656539e+00
  1.37540436e+00  1.31817261e+00  1.28412366e+00  1.28110203e+00
  1.20402336e+00  1.19178252e+00  1.14189239e+00  1.13771501e+00
  1.06909976e+00  1.03098386e+00  1.02580668e+00  1.01818840e+00
  9.64594907e-01  9.49221779e-01  9.40827140e-01  8.65292423e-01
  8.33410723e-01  7.59887872e-01  7.06956551e-01  6.71713053e-01
  6.51621966e-01  5.98395331e-01  5.72437104e-01  5.43552737e-01
  5.07196468e-01  4.81243121e-01  4.66532627e-01  4.53475343e-01
  4.50563936e-01  3.75915711e-01  3.46834138e-01  3.28634009e-01
  3.24011605e-01  2.57842868e-01  2.20946571e-01  2.00874456e-01
  1.25771435e-01  5.30462252e-02  4.06924911e-02  1.02103276e-02
 -9.33899786e-03 -3.77809826e-02 -6.46489334e-02 -9.85042693e-02
 -1.44146734e-01 -1.86019059e-01 -1.96823288e-01 -2.03347346e-01
 -2.33473764e-01 -2.57661313e-01 -3.17788412e-01 -3.31169146e-01
 -3.45713856e-01 -3.78959235e-01 -4.12969391e-01 -4.26620415e-01
 -4.46871679e-01 -4.78871977e-01 -5.24949218e-01 -5.40216362e-01
```

```
-5.46630884e-01 -5.59217420e-01 -5.66433738e-01 -5.95596227e-01
-6.22262842e-01 -6.85114425e-01 -6.97161297e-01 -7.21601787e-01
-7.34020617e-01 -7.41850765e-01 -7.57152652e-01 -7.62883373e-01
-7.72596705e-01 -7.79404341e-01 -7.90204359e-01 -8.00309208e-01
-8.10601429e-01 -8.18975695e-01 -8.20119027e-01 -8.21657658e-01
-8.28961412e-01 -8.29768540e-01 -8.29987337e-01 -8.32478278e-01
-8.31555101e-01 -8.31200533e-01 -8.30839183e-01 -8.28770949e-01
-8.22776619e-01 -8.21828948e-01 -8.16254887e-01 -8.13579087e-01
-8.13405158e-01 -8.06042461e-01 -8.05695711e-01 -8.02372210e-01
-7.69680306e-01 -7.57657283e-01 -7.57254870e-01 -7.49566682e-01
-7.42650293e-01 -7.40517323e-01 -7.36183865e-01 -7.22623103e-01
-7.19066807e-01 -7.13108733e-01 -7.04095964e-01 -6.83297159e-01
-6.80046372e-01 -6.54751752e-01 -6.36362965e-01 -6.25905219e-01
-6.20113324e-01 -6.17765909e-01 -5.96118063e-01 -5.64330471e-01
-5.55970468e-01 -4.96377648e-01 -4.86856798e-01 -4.83736805e-01
-4.57358434e-01 -4.41391409e-01 -4.18077018e-01 -3.94502868e-01
-3.89888199e-01 -3.83310075e-01 -3.32548117e-01 -2.88934333e-01
-2.79251616e-01 -2.62416136e-01 -1.72809586e-01 -1.43748812e-01
-9.81767233e-02 -7.96260883e-03 -3.38408146e-04  9.22479451e-02
 1.04581939e-01  1.33992084e-01  1.61531218e-01  1.99404696e-01
 2.30108683e-01  2.69406862e-01  3.01427936e-01  3.26599403e-01
 3.77181458e-01  4.53049772e-01  4.88429961e-01  5.19801186e-01
 5.89763951e-01  6.39459076e-01  6.93532738e-01  6.98741009e-01
 7.17425046e-01  7.72487948e-01  8.92988679e-01  9.57143296e-01
 9.69704138e-01  9.84220557e-01  1.00770341e+00  1.03675353e+00
 1.04195204e+00  1.05577522e+00  1.13963015e+00  1.15053016e+00
 1.23558431e+00  1.24505835e+00  1.30266391e+00  1.32361197e+00
 1.42116714e+00  1.47165004e+00  1.48023403e+00  1.49035304e+00
 1.54200210e+00  1.56065709e+00  1.57203517e+00  1.61876614e+00
 1.70496131e+00  1.74918480e+00  1.75977057e+00  1.80146805e+00
 1.87448822e+00  1.89185783e+00  1.94008562e+00  1.96360809e+00]
```

[50]:
```
###### plotting the gradient norm vs number of itertaions curve

fig1 = plt.figure(figsize = (8,8))
x_axis = list(range(counter-1))
plt.plot(x_axis[1500:], G_norm[1500:])
plt.xlabel('Number of iterations')
plt.ylabel('Norm of gradient')
plt.title('gradient norm vs no. of iterations after 1500 iterations')
plt.show()
```



6 Question-4

Compare the obtained solution by the gradient and stochastic gradient algorithm with the solution obtained by the direct method (solving the normal equation) in each case.

[51]:
```
M_list = np.array([1,2,5])
lamb = 0.001
values_in_eq = (M_list+1) * (M_list+2)/2
RMSE_RPR_normal = []

for i in range(len(values_in_eq)):
  M = M_list[i]
  val = values_in_eq[i]
```

```
 x_matrix_bivariate = generate_bivariate_x_matrix(train_X1, train_X2, M)
 w_optimal_bivariate = np.linalg.inv((x_matrix_bivariate.T @␣
↪x_matrix_bivariate) + lamb * np.identity(int(val))) @ x_matrix_bivariate.T @␣
↪train_Y
 # # w_matrix_bivariate.append(w_optimal_bivariate)
 # y_est_bivariate = w_optimal_bivariate @ (x_matrix_bivariate.T)
 # RMSE_bivariate = calculate_RMSE()
 print('For M value : '+str(M))
 # print('Training RMSE : '+str(RMSE_bivariate))

 x_matrix_bivariate_test = generate_bivariate_x_matrix(test_X1, test_X2, M)
 y_est_bivariate_test = w_optimal_bivariate @ (x_matrix_bivariate_test.T)
 RMSE_bivariate_test = calculate_RMSE(y_est_bivariate_test, test_Y)
 print('Testing RMSE : '+str(RMSE_bivariate_test))
␣
↪print('================================================================================')
 RMSE_RPR_normal.append(RMSE_bivariate_test)
```

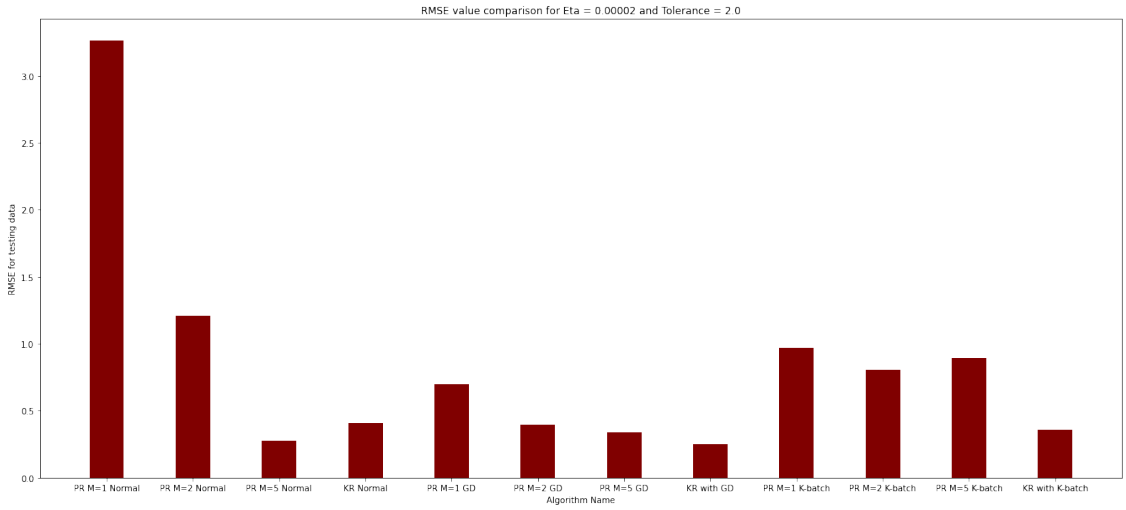
```
For M value : 1
Testing RMSE : 3.261510601840822
================================================================================
For M value : 2
Testing RMSE : 1.2100156017403978
================================================================================
For M value : 5
Testing RMSE : 0.27889741280983854
================================================================================
```

[52]:
```
RMSE_RPR_normal
```

[52]:
```
[3.261510601840822, 1.2100156017403978, 0.27889741280983854]
```

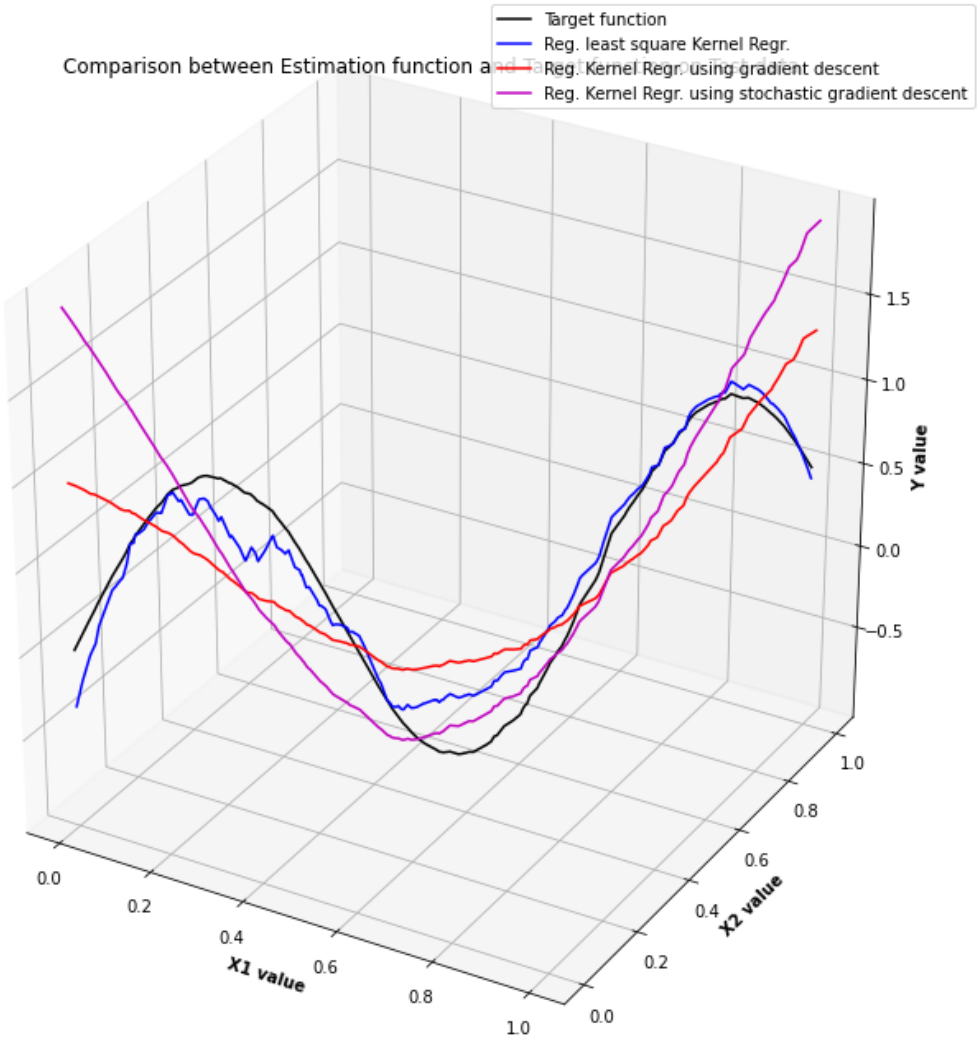
[53]:
```
x_axis = ['PR M=1 Normal', 'PR M=2 Normal', 'PR M=5 Normal', 'KR Normal', 'PR␣
 ↪M=1 GD', 'PR M=2 GD', 'PR M=5 GD', 'KR with GD', 'PR M=1 K-batch', 'PR M=2␣
 ↪K-batch', 'PR M=5 K-batch', 'KR with K-batch']
y_axis = RMSE_RPR_normal + [test_RMSE_RKR_Q1] + RMSE_RPR + [test_RMSE_RKR_Q2] +␣
 ↪RMSE_RPR_K_batch + [test_RMSE_RKR_Q3]

fig2 = plt.figure(figsize = (23,10))

# creating the bar plot
plt.bar(x_axis, y_axis, color ='maroon', width = 0.4)

plt.xlabel("Algorithm Name")
plt.ylabel("RMSE for testing data")
plt.title("RMSE value comparison for Eta = 0.00002 and Tolerance = 2.0")
```

```
plt.show()
```



[71]:
```
def actual_values(X1, X2):
  PI = math.pi
  Exp = 2 * PI * ((X1**2 + X2**2) ** 0.5)
  actual_Z = np.sin(Exp)
  return actual_Z
```

[75]:
```
Z = actual_values(test_X1, test_X2)

fig1 = plt.figure(figsize = (12,12))

ax = plt.axes(projection ="3d")
ax.plot(test_X1, test_X2, Z, color = "k", label='Target function')
ax.plot(test_X1, test_X2, test_Y_est, color = "blue", label='Reg. least square␣
 ↪Kernel Regr.')
ax.plot(test_X1, test_X2, test_Y_est_Q2, color = "r", label='Reg. Kernel Regr.␣
 ↪using gradient descent')
ax.plot(test_X1, test_X2, test_Y_est_Q3, color = "m", label='Reg. Kernel Regr.␣
 ↪using stochastic gradient descent');

plt.title("Comparison between Estimation function and Target function on Test␣
 ↪data")
ax.set_xlabel('X1 value', fontweight ='bold')
ax.set_ylabel('X2 value', fontweight ='bold')
ax.set_zlabel('Y value', fontweight ='bold')
ax.legend()
plt.show()
```



7 Question-5

Study the behavior of the chosen step length eta regarding the convergence to actual solution in case of all used algorithms.

7.1 For eta = 0.00002

[ ]:
```
M_list = np.array([1,2,5])
values_in_eq = (M_list+1) * (M_list+2)/2
RMSE_RPR = []
iterations = []

lamb = math.exp(-18)
```

```
eta = 0.00002
tolerance = 2.0

for i in range(len(values_in_eq)):
  M = M_list[i]
  val = values_in_eq[i]

  x_matrix_bivariate = generate_bivariate_x_matrix(train_X1, train_X2, M)
  x_matrix_bivariate_test = generate_bivariate_x_matrix(test_X1, test_X2, M)
  weight_coeff = np.array([1]*len(x_matrix_bivariate[0]))

  counter = 1
  G_norm = []

  G = compute_gradient(lamb, weight_coeff, x_matrix_bivariate, train_Y)

  while np.linalg.norm(G) > tolerance:
    weight_coeff = weight_coeff - eta * G
    G_norm.append(np.linalg.norm(G))
    G = compute_gradient(lamb, weight_coeff, x_matrix_bivariate, train_Y)

    print('Iteration no : '+str(counter))
    counter = counter+1
    print('Current gradient\'s norm value is : '+str(np.linalg.norm(G)))
    print('==================================================================')

  test_Y_est_RPR = x_matrix_bivariate_test @ weight_coeff
  RMSE_current_M = calculate_RMSE(test_Y_est_RPR, test_Y)

  print('Testing RMSE for M={} is : {}'.format(M, RMSE_current_M))
  print('Total number of iterations performed to get the values less then the␣
 ↪tolerance are : '+str(counter-1))
 ␣
 ↪print('================================================================================')

  RMSE_RPR.append(RMSE_current_M)
  iterations.append(counter-1)
```

[55]:
```
print('Testing RMSE values for M = 1,2 and 5 are respectively : {}'.
 ↪format(RMSE_RPR))
print('Iterations taken for M = 1,2 and 5 are respectively : {}'.
 ↪format(iterations))
```

```
Testing RMSE values for M = 1,2 and 5 are respectively : [0.6988961589749403,
0.3949235200352556, 0.3362408908496466]
Iterations taken for M = 1,2 and 5 are respectively : [48055, 40376, 65320]
```

7.2 For eta = 0.0002

[ ]:
```
M_list = np.array([1,2,5])
values_in_eq = (M_list+1) * (M_list+2)/2
RMSE_RPR = []
iterations = []

lamb = math.exp(-18)
eta = 0.0002
tolerance = 2.0

for i in range(len(values_in_eq)):
  M = M_list[i]
  val = values_in_eq[i]

  x_matrix_bivariate = generate_bivariate_x_matrix(train_X1, train_X2, M)
  x_matrix_bivariate_test = generate_bivariate_x_matrix(test_X1, test_X2, M)
  weight_coeff = np.array([1]*len(x_matrix_bivariate[0]))

  counter = 1
  G_norm = []

  G = compute_gradient(lamb, weight_coeff, x_matrix_bivariate, train_Y)

  while np.linalg.norm(G) > tolerance:
    weight_coeff = weight_coeff - eta * G
    G_norm.append(np.linalg.norm(G))
    G = compute_gradient(lamb, weight_coeff, x_matrix_bivariate, train_Y)

    print('Iteration no : '+str(counter))
    counter = counter+1
    print('Current gradient\'s norm value is : '+str(np.linalg.norm(G)))
    print('================================================================')

  test_Y_est_RPR = x_matrix_bivariate_test @ weight_coeff
  RMSE_current_M = calculate_RMSE(test_Y_est_RPR, test_Y)

  print('Testing RMSE for M={} is : {}'.format(M, RMSE_current_M))
  print('Total number of iterations performed to get the values less then the␣
↪tolerance are : '+str(counter-1))
␣
↪print('================================================================================')

  RMSE_RPR.append(RMSE_current_M)
  iterations.append(counter-1)
```

[57]:
```
print('Testing RMSE values for M = 1,2 and 5 are respectively : {}'.
 ↪format(RMSE_RPR))
print('Iterations taken for M = 1,2 and 5 are respectively : {}'.
 ↪format(iterations))
```

```
Testing RMSE values for M = 1,2 and 5 are respectively : [0.6989026841176795,
0.39491741386577334, 0.3362403894824575]
Iterations taken for M = 1,2 and 5 are respectively : [4806, 4037, 6532]
```

7.3 For eta = 0.002

[ ]:
```
M_list = np.array([1,2,5])
values_in_eq = (M_list+1) * (M_list+2)/2
RMSE_RPR = []
iterations = []

lamb = math.exp(-18)
eta = 0.002
tolerance = 2.0

for i in range(len(values_in_eq)):
  M = M_list[i]
  val = values_in_eq[i]

  x_matrix_bivariate = generate_bivariate_x_matrix(train_X1, train_X2, M)
  x_matrix_bivariate_test = generate_bivariate_x_matrix(test_X1, test_X2, M)
  weight_coeff = np.array([1]*len(x_matrix_bivariate[0]))

  counter = 1
  G_norm = []

  G = compute_gradient(lamb, weight_coeff, x_matrix_bivariate, train_Y)

  while np.linalg.norm(G) > tolerance:
    weight_coeff = weight_coeff - eta * G
    G_norm.append(np.linalg.norm(G))
    G = compute_gradient(lamb, weight_coeff, x_matrix_bivariate, train_Y)

    print('Iteration no : '+str(counter))
    counter = counter+1
    print('Current gradient\'s norm value is : '+str(np.linalg.norm(G)))
    print('==============================================================')

  test_Y_est_RPR = x_matrix_bivariate_test @ weight_coeff
  RMSE_current_M = calculate_RMSE(test_Y_est_RPR, test_Y)

  print('Testing RMSE for M={} is : {}'.format(M, RMSE_current_M))
```

```
 print('Total number of iterations performed to get the values less then the␣
↪tolerance are : '+str(counter-1))
␣
↪print('================================================================================')

 RMSE_RPR.append(RMSE_current_M)
 iterations.append(counter-1)
```

[59]:
```
print('Testing RMSE values for M = 1,2 and 5 are respectively : {}'.
↪format(RMSE_RPR))
print('Iterations taken for M = 1,2 and 5 are respectively : {}'.
↪format(iterations))
```

```
Testing RMSE values for M = 1,2 and 5 are respectively : [0.6989554462813905,
0.39486050593360966, nan]
Iterations taken for M = 1,2 and 5 are respectively : [481, 403, 1139]
```

8 Thank You!

[77]:
```
!wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
from colab_pdf import colab_pdf
colab_pdf('PRML03.ipynb')
```

```
File colab_pdf.py already there; not retrieving.


WARNING: apt does not have a stable CLI interface. Use with caution in scripts.


WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

[NbConvertApp] Converting notebook /content/drive/MyDrive/Colab
Notebooks/PRML03.ipynb to pdf
[NbConvertApp] Support files will be in PRML03_files/
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
[NbConvertApp] Making directory ./PRML03_files
```

```
[NbConvertApp] Writing 122394 bytes to ./notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: [u'xelatex', u'./notebook.tex',
'-quiet']
[NbConvertApp] Running bibtex 1 time: [u'bibtex', u'./notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 446300 bytes to /content/drive/My Drive/PRML03.pdf

<IPython.core.display.Javascript object>


<IPython.core.display.Javascript object>
```

[77]:
```
'File ready to be Downloaded and Saved to Drive'
```

[ ]:
```

```