1003243509 Agia 1
February 28, 2019

ROB313 Intro to Learning from Data - Assignment 2

Objectives

The objective of the assignment is to gain experience implementing Generalized Linear Models (GLM's) in both standard and kernelized forms. This requires an understanding of basis functions and how they can be constructed for accurate prediction on both regression and classification sets. By using the same datasets as assignment 1, we will be able to compare the results acquired from the GLM to the results from the regular Linear Model and the k-NN Algorithm, and analyze the effect of introducing non-linearity in our inputs to our model. Furthermore, we compare several characteristics (e.g. Time and Space) of a standard GLM to the Kernelized GLM.

Code Structure and Strategies

The code structure is designed to be modular and easily understandable. Each question has a corresponding code section in the script and in the main block, which is boldly outlined with comments. To run the code for each question, the only modifications needed to be made to the file is to uncomment the corresponding section outlined in the main block. The results of each question are printed neatly on the terminal when run, and relevant plots will be generated and saved in the current directory. The remaining code is structured as follows:

Q1: **glm_validation(**data**)**, **glm_test(**data, regularization parameter**)**

In question 1, the **glm_validation()** function is used to determine the optimal regularization parameter by making predictions for the mauna_loa dataset and minimizing the RMSE value on the validation set. The optimal regularization parameter is then passed into **glm_test()** which then computes test predictions on the mauna_loa test set, and returns the test RMSE value.

Q2: **glm_kernelized(**data, regularization parameter**)**, **visualize_kernel()**

A kernelized form of the GLM in question one is implemented in **glm_kernelized()**, which makes predictions on the mauna_loa test set using the optimal regularization parameter obtained by **glm_validation()**. Plots of the kernel are generated by **visualize_kernel()**.

Q3: **gaussian_rbf_glm_valid(**data**)**, **gaussian_rbf_glm_test**(data, regularization param, theta**)**

A kernelized GLM is implemented using a gaussian radial basis function for both regression and classification. For each dataset, **gaussian_rbf_glm_valid()** is used to make predictions for the specified lengthscale values and regularization values. The optimal regularization and lengthscale value is then passed to **gaussian_rbf_glm_test()** to compute predictions on the test sets, and return the test RMSE (regression) or test accuracy ratio (classification).

Shared Functions:

- **custom_kernel(**x, z**)** – returns the kernelized inner product of the basis functions of x and z
- **custom_vector(**x**)** – returns a custom basis function vector of value x

Commenting and code organization were used to make the code easy to read and to reduce the time spent searching for critical/error prone sections. Utility functions for frequently used functions (e.g. root mean squared error) were written. The other functions were designed to be modular so that they can be implemented to make predictions on datasets with varying feature space dimensions. When designing the customized basis function for question 1, the mauna_loa dataset was plotted to identify macro and micro trends in the data. This was an essential step to improving the performance of the model. The custom basis function was also carefully constructed to that it could be kernelized, and used to construct the kernelized GLM in question 2. Question 3 required generating predictions on several datasets with a gaussian RBF kernel, with varying values for the shape parameter and regularization parameter. Thus, a loop structure that optimized the usage of re-usable matrices/vectors was deployed to reduce the computational complexity of the predictions.

Q1 – GLM with Customized Basis Function (Primal Approach)

A generalized linear model was implemented to make predictions on the one-dimensional mauna_loa dataset. The optimal regularization (λ) parameter was selected by looping over values $\lambda \in [0, 30]$ and selecting the regularization parameter that minimized the RMSE error on the validation set. This regularization parameter was then used to compute the predictions on the test set, and the test RMSE was computed. The performance of the model is shown in Table 1.

Table 1 – Results of GLM on the mauna_loa Dataset

Dataset	$\boldsymbol{\lambda_{optimal}}$	**Test RMSE**
Mauna Loa	14	0.103937949238

The model was constructed using the basis function mapping the one-dimensional dataset to five-dimensional feature space:

$$\boldsymbol{\phi}(x)\colon \mathbb{R} \to \mathbb{R}^5, \boldsymbol{\phi}(x) = \left[1, \sqrt{2}x, x^2, xsin(\omega x), xcos(\omega x)\right]^T$$

The angular frequency $\omega = \frac{2\pi}{0.0565}$ was determined empirically by plotting the mauna_loa training set and computing the period of oscillation of the macro trend. The basis function was designed to fit both the macro and micro trends of the mauna_loa dataset. The macro trend of the dataset displayed characteristics of a linearly increasing sinusoid, with micro trends that could be approximated with linear and quadratic terms. The original basis function included an x^3 term to approximate cubic micro trends, however, the associated reduction in error was not significant enough to justify its inclusion in the basis function. With the above basis function, the performance of the model was quite impressive with a Test RMSE at approximately 10% with a regularization parameter equal to 14 (i.e. significant improvement to regular Linear Model). The predictions on the test set are shown in Figure 1.

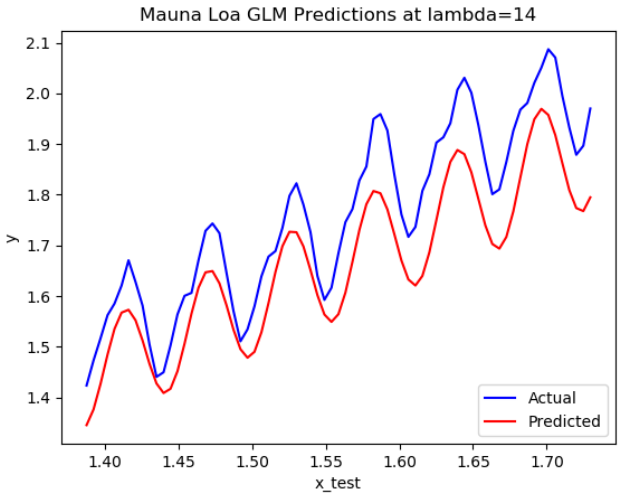

Figure 1 – GLM Predictions on Mauna Loa Test Set

Q2 – Kernelized GLM (Dual Perspective)

A kernelized form the GLM from question one was constructed from the dual perspective. This process involved deriving a kernel function representing fast inner products of the basis function shown in question 1. The kernel was simplified to the following form:

$$k(x,z) = \boldsymbol{\phi}(x)^T \boldsymbol{\phi}(z) = (1 + x * z)^2 + \cos(\omega(x - z))$$

The predictions to the mauna_loa test set are shown in Figure 2, and the Test RMSE value with a regularization parameter of 14 in shown in Table 2. As expected, the results are identical to the primal GLM as the models are numerically equivalent, but expressed in different forms.

Table 2 – Results of Kernelized GLM on the mauna_loa Dataset

Dataset	$\boldsymbol{\lambda_{optimal}}$	**Test RMSE**
Mauna Loa	14	0.103937949238

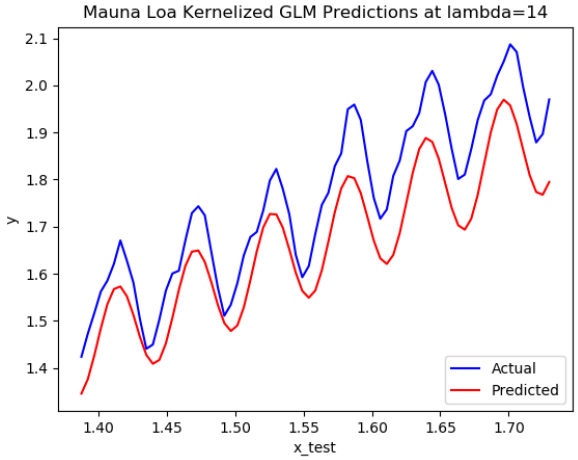

Figure 2 – Kernelized GLM Predictions on Mauna Loa Test Set

In this case, the kernelized GLM is actually worse in terms of both computational cost and memory requirements. The Gram matrix used in the kernelized GLM dominates the memory requirements for the model, making it $O(N^2)$. The standard GLM memory requirements is bounded by $O(N * M)$ which comes from storing the Φ matrix which consists of the basis function for all training points. In terms to computational cost, the kernelized GLM is bounded $O(N^3)$, while the SVD used in the standard GLM dominates the computational cost with $O(2N(M+1)^2 + 11N(M+1)^3)$. In our case, $M = 5$ but N is a significantly larger number. Since the computational cost of the kernelized model is proportional the N^3, while the standard GLM is only linearly proportional to N (i.e. $(M+1)^3$ is almost negligible compared N^2), it is evident that the standard GLM is computationally more efficient for the designed basis function.

The kernel was analyzed by developing two plots. Each graph plots the kernel at a specific value of x across a range of z values such that $z \in [-0.1, 0.1]$. The plots are shown in Figures 3-4.

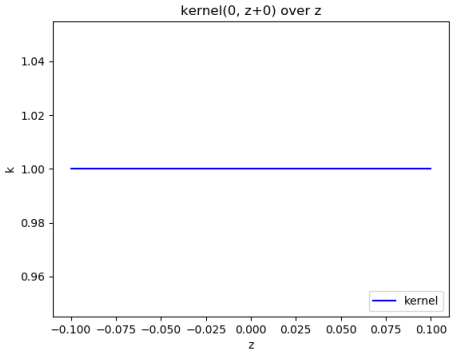

Figure 3 – k(0, z), $z \in [-0.1, 0.1]$

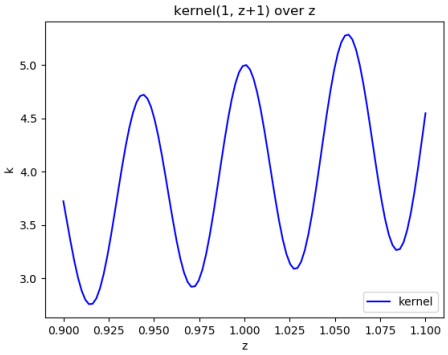

Figure 4 – k(1, z+1), $z \in [-0.1, 0.1]$

Analyzing the above figures, it is evident that the kernel is not translationally invariant. The kernel is shown depend on more than just the numerical difference between datapoint x and points z. As we can see, adding a constant value of $i = 1$ to $k(0 + i, z + i)$ provides a very different result in Figure 4 in comparison to Figure 3.

Q3 – Gaussian RBF Model for Regression and Classification

A GLM model using a gaussian RBF kernel was constructed to preform regression on both the mauna_loa and rosenbrock datasets, and classification on the iris dataset. For each dataset, the regularization parameter and length scale pair that minimized validation RMSE (regression) or maximized the validation accuracy ratio (classification) was acquired and used to make predictions on the test sets. Table 3 summarizes the results for each dataset.

Table 3 – Gaussian RBF GLM Results on Regression and Classification Sets

Dataset	**Data Type**	$\boldsymbol{\lambda_{optimal}}$	$\boldsymbol{\theta_{optimal}}$	**Validation RMSE/Ratio**	**Test RMSE/Ratio**
Mauna Loa	Regression	0.001	1	0.124478670316	0.149773387722
Rosenbrock	Regression	0.001	2	0.193239586974	0.14812442755
Iris	Classification	1	0.5	0.870967741935	1.0

Q4 – Tikhonov Regularization

$$\underset{w \in \mathbb{R}^D}{\operatorname{argmin}} \|y - Xw\|_2^2 + w^T \Gamma w \quad \text{where } \Gamma \text{ is symmetric SPD matrix.}$$

$$\nabla_w \left[(y - Xw)^T (y - Xw) + w^T \Gamma w \right] = 0$$

$$= 2X^T X w - 2X^T y + 2\Gamma w = 0$$

Therefore, $(X^T X + \Gamma) w = X^T y$

and finally, $\boxed{w = (X^T X + \Gamma)^{-1} X^T y}$

Q5 – Weight Estimation comparison to the Dual Perspective

Objective function for model $f(x, \alpha) = \sum_{i=1}^{N} \alpha_i k(x, x^{(i)}) = K^T \alpha$

is given as $\sum_{i=1}^{N} (y^{(i)} - f(x^{(i)}, \alpha))^2 + \lambda \sum_{i=1}^{N} \alpha_i^2$

Kernels, inner products of basis functions

Recast into matrix form as: $\|y - K\alpha\|_2^2 + \lambda \|\alpha\|_2^2$ where $K = \Phi \Phi^T$ is the Gram Matrix

Now, $\nabla_\alpha \left[(y - K\alpha)^T (y - K\alpha) + \lambda \alpha^T \alpha \right] = 0$

$$= 2K^T K \alpha - 2K^T y + 2\lambda \alpha$$

Therefore, $(K^T K + \lambda I) \alpha = K^T y$

and finally, $\boxed{\alpha = (K^T K + \lambda I)^{-1} K^T y}$

It is clear that this result is different than the expression derived in class. The previous expression, (1) $\alpha = (K + \lambda I)^{-1} y$, was derived by using the Matrix Inversion Lemma to transform $\omega = (\Phi^T \Phi + \lambda I)^{-1} \Phi^T y$ into a form that kernelized the model with fast inner products between basis functions. In this question, the box result was derived by minimizing the L2-loss of a pre-kernelized model. A proof below is presented to support this argument.

Proof: Let $\underline{\alpha}' = (\underline{\underline{K}}^T \underline{\underline{K}} + \lambda \underline{\underline{1}})^{-1} \underline{\underline{K}}^T \underline{y}$ (result of minimization of objective function)

and $\underline{\alpha} = (\underline{\underline{K}} + \lambda \underline{\underline{1}})^{-1} \underline{y}$ (dual perspective result)

Set $\underline{\alpha} = \underline{\alpha}' \Longrightarrow (\underline{\underline{K}}^T \underline{\underline{K}} + \lambda \underline{\underline{1}})^{-1} \underline{\underline{K}}^T \underline{y} = (\underline{\underline{K}} + \lambda \underline{\underline{1}})^{-1} \underline{y}$

$$\underline{\underline{K}}^T (\underline{\underline{K}} + \lambda \underline{\underline{1}}) = \underline{\underline{K}}^T \underline{\underline{K}} + \lambda \underline{\underline{1}}$$

$$\lambda \underline{\underline{K}}^T = \lambda \underline{\underline{1}}$$

Thus, $\underline{\alpha} = \underline{\alpha}'$ implies that $\boxed{\underline{\underline{K}}^T = \underline{\underline{1}}}$, which is not true.

The i^{th}, j^{th} element in $\underline{\underline{K}}$ is $k(\underline{x}^{(i)}, \underline{x}^{(j)})$.

① Need $K(\underline{x}^{(i)}, \underline{x}^{(j)}) = 0$ for $i \neq j$

② and $k(\underline{x}^{(i)}, \underline{x}^{(j)}) = 0$ for $i = j$

} Necessary for $\underline{\underline{K}} = \underline{\underline{1}}$

However, conditions ① and ② are not true for majority of basis functions and kernels.

Therefore, we have a conflict.

Therefore, $\underline{\alpha} \neq \underline{\alpha}'$

```python
import numpy as np
import math
from matplotlib import pyplot as plt
from data_utils import load_dataset


__author__ = 'Christopher Agia (1003243509)'
__date__ = 'February 18, 2019'


# Useful Utility Functions
def compute_rmse(y_test, y_estimates):
    return np.sqrt(np.average((y_test-y_estimates)**2))


def l2_norm(x1, x2):
    return np.linalg.norm([x1-x2], ord=2)


# Kernel Functions and Custom Vector Function
def custom_kernel(x0, x1):
    # Customer kernel incorporates a polynomial expansion term and product of two sinusoidal terms
    return (1+x0*x1)**2 + x0*x1*math.cos(2*math.pi/0.0565*(x0-x1))


def custom_vector(x):
    freq = 2*math.pi/0.0565
    row = list()
    row.append(1)
    row.append(math.sqrt(2)*x)
    row.append(x**2)
    row.append(x * math.sin(freq * x))
    row.append(x * math.cos(freq * x))
    return np.array(row)


# ---------------------- Question 1 ----------------------

def glm_validation(x_train, x_valid, y_train, y_valid, l_vals=None):

    # Regularization Parameters ranging from 0-30
    if not l_vals:
        l_vals = list(range(0, 31))

    # Create and Populate the PHI Matrix
    shape = (len(x_train), 5)
    phi = np.empty(shape)
    for i in range(len(x_train)):
        phi[i, :] = custom_vector(x_train[i])

    # Create validation phi matrix
    shape = (len(x_valid), 5)
    phi_valid = np.empty(shape)
    for i in range(len(x_valid)):
        phi_valid[i, :] = custom_vector(x_valid[i])

    # Compute SVD
    U, S, Vh = np.linalg.svd(phi)

    # Invert Sigma
    sig = np.diag(S)
    filler = np.zeros([len(x_train) - len(S), len(S)])
    sig = np.vstack([sig, filler])

    # Compute weights and predictions with varying lambda values
    min_rmse = np.inf
    for l_val in l_vals:
        temp0 = np.dot(sig.T, sig)
        temp1 = np.linalg.inv(temp0 + l_val*np.eye(len(temp0)))

        w = np.dot(Vh.T, np.dot(temp1, np.dot(sig.T, np.dot(U.T, y_train))))

        predictions = np.dot(phi_valid, w)
```

```
74          rmse_val = compute_rmse(y_valid, predictions)
75
76          if rmse_val < min_rmse:
77              min_rmse = rmse_val
78              l_min = l_val
79
80      return l_min
81
82
83  def glm_test(x_train, x_valid, x_test, y_train, y_valid, y_test, l_val):
84
85      x_total = np.vstack([x_train, x_valid])
86      y_total = np.vstack([y_train, y_valid])
87
88      # Create and Populate the training PHI Matrix
89      shape = (len(x_total), 5)
90      phi = np.empty(shape)
91      for i in range(len(x_total)):
92          phi[i, :] = custom_vector(x_total[i])
93
94      # Create test PHI matrix
95      shape = (len(x_test), 5)
96      phi_test = np.empty(shape)
97      for i in range(len(x_test)):
98          phi_test[i, :] = custom_vector(x_test[i])
99
100     # Compute SVD
101     U, S, Vh = np.linalg.svd(phi)
102
103     # Invert Sigma
104     sig = np.diag(S)
105     filler = np.zeros([len(x_total) - len(S), len(S)])
106     sig = np.vstack([sig, filler])
107
108     # Compute Test Predictions
109     temp0 = np.dot(sig.T, sig)
110     temp1 = np.linalg.inv(temp0 + l_val * np.eye(len(temp0)))
111     w = np.dot(Vh.T, np.dot(temp1, np.dot(sig.T, np.dot(U.T, y_total))))
112     predictions = np.dot(phi_test, w)
113
114     test_error = compute_rmse(y_test, predictions)
115
116     plt.figure(1)
117     plt.plot(x_test, y_test, '-b', label='Actual')
118     plt.plot(x_test, predictions, '-r', label='Predicted')
119     plt.title('Mauna Loa GLM Predictions at lambda=' + str(l_val))
120     plt.xlabel('x_test')
121     plt.ylabel('y')
122     plt.legend(loc='lower right')
123     plt.savefig('mauna_loa_glm_estimates.png')
124
125     return test_error
126
127
128 # ---------------------- Question 2 ----------------------
129
130 def glm_kernelized(x_train, x_valid, x_test, y_train, y_valid, y_test, l_val):
131
132     x_total = np.vstack([x_train, x_valid])
133     y_total = np.vstack([y_train, y_valid])
134
135     # Create and Populate the Gram Matrix (K)
136     shape = (len(x_total), len(x_total))
137     K = np.empty(shape)
138     prev_computed = {}                          # Stores previously computed custom kernels
139     for i in range(len(x_total)):
140         for j in range(len(x_total)):
141             a = x_total[i]
142             b = x_total[j]
143             # Add to previously computed dictionary
144             if str((a, b)) not in prev_computed:
145                 prev_computed[str((a, b))] = custom_kernel(a, b)
146                 prev_computed[str((b, a))] = prev_computed[str((a, b))]
147             # Add kernel to K (Gram) Matrix
```

```
148             K[i, j] = prev_computed[str((a, b))]
149 
150     # Cholesky Factorization of K + lambda*1, here R is lower triangular
151     R = np.linalg.cholesky((K + l_val*np.eye(len(K))))
152 
153     # Find inverse, P = inv(R) makes it quicker to find matrix inverse
154     P = np.linalg.inv(R)
155 
156     # Estimate dual-variables alpha
157     alp = np.dot(np.dot(P.T, P), y_total)
158 
159     # Compute predictions
160     predictions = np.empty(np.shape(y_test))
161     for i in range(len(x_test)):
162         # Create k vector, containing the kernel products of x_test and all x_training points
163         k = np.empty(np.shape(alp))
164         for j in range(len(x_total)):
165             k[j] = custom_kernel(x_test[i], x_total[j])
166         # Make prediction for x_test[i]
167         predictions[i] = np.dot(k.T, alp)
168 
169     # Compute model test_error
170     test_error = compute_rmse(y_test, predictions)
171 
172     plt.figure(2)
173     plt.plot(x_test, y_test, '-b', label='Actual')
174     plt.plot(x_test, predictions, '-r', label='Predicted')
175     plt.title('Mauna Loa Kernelized GLM Predictions at lambda=' + str(l_val))
176     plt.xlabel('x_test')
177     plt.ylabel('y')
178     plt.legend(loc='lower right')
179     plt.savefig('mauna_loa_glm_kernel_estimates.png')
180 
181     return test_error
182 
183 
184 def visualize_kernel():
185 
186     for i in range(2):
187         y_vals = list()
188 
189         z = np.linspace(-0.1 + i, 0.1 + i, 100)
190         z = np.array(z)
191 
192         for elem in z:
193             y_vals.append(custom_kernel(i, elem))
194 
195         plt.figure(i + 3)
196         plt.plot(z, y_vals, '-b', label='kernel')
197         plt.title('kernel(' + str(i) + ', z+' + str(i) + ') over z')
198         plt.xlabel('z')
199         plt.ylabel('k')
200         plt.legend(loc='lower right')
201         plt.savefig('kernel' + str(i) + '.png')
202 
203 
204 # ---------------------- Question 3 ----------------------
205 
206 def gaussian_rbf_glm_valid(x_train, x_valid, y_train, y_valid, dataset):
207 
208     theta_vals = [0.05, 0.1, 0.5, 1, 2]
209     reg_vals = [0.001, 0.01, 0.1, 1]
210     # Will store the validation_errors/validation_accuracy for each theta-regularization value pair
211     results = {}
212 
213     # theta value only affects Gram Matrix and K Vector used in prediction
214     for theta in theta_vals:
215 
216         # Create and Populate the Gram Matrix (K) at the current theta value
217         shape = (len(x_train), len(x_train))
218         K = np.empty(shape)                 # Gram Matrix
219         prev_computed = {}                  # Store previously computed gaussian kernel
220         for i in range(len(x_train)):
221             for j in range(len(x_train)):
```

```
222                 a = x_train[i]
223                 b = x_train[j]
224                 if str((a, b)) not in prev_computed:
225                     prev_computed[str((a, b))] = gaussian_rbf(a, b, theta)
226                     prev_computed[str((b, a))] = prev_computed[str((a, b))]
227                 # Add kernel to K (Gram) Matrix
228                 K[i, j] = prev_computed[str((a, b))]
229 
230         # Only alpha changes with regularization value, thus calculate k_vector (and extend to matrix form = kM)
231         kM = np.empty((len(x_valid), len(x_train)))
232         for i in range(len(x_valid)):
233             # Create k vector, containing the kernel products of x_test and all x_training points
234             k = list()
235             vec = x_valid[i]
236             for j in range(len(x_train)):
237                 k.append(gaussian_rbf(vec, x_train[j], theta))
238             kM[i, :] = np.array(k)
239 
240         # K, kM matrix computed, compute test_errors for each regularization value at current theta
241         for l_val in reg_vals:
242 
243             # Cholesky Factorization of K + lambda*1, here R is lower triangular
244             R = np.linalg.cholesky((K + l_val * np.eye(len(K))))
245             # Find inverse, P = inv(R) makes it quicker to find matrix inverse
246             P = np.linalg.inv(R)
247             # Estimate dual-variables alpha
248             alp = np.dot(np.dot(P.T, P), y_train)
249 
250             # Compute Test RMSE for regresion datasets
251             if dataset == 'mauna_loa' or dataset == 'rosenbrock':
252                 # Compute predictions
253                 predictions = np.dot(kM, alp)
254                 # Compute model validation error at current regularization-theta pair, and store in results
255                 results[(theta, l_val)] = compute_rmse(y_valid, predictions)
256 
257             # Compute Test Accuracy Ratio for classification datasets
258             else:
259                 # Compute predictions
260                 predictions = np.argmax(np.dot(kM, alp), axis=1)
261                 y_valid0 = np.argmax(1 * y_valid, axis=1)
262                 # Compute model prediction accuracy at current regularization-theta pair, and store in results
263                 results[(theta, l_val)] = (predictions == y_valid0).sum() / len(y_valid0)
264 
265     # Acquire the optimal theta and regularization values, return them to be used for test set
266     if dataset == 'mauna_loa' or dataset == 'rosenbrock':
267         opt_res = np.inf
268         for theta, l_val in results:
269             if results[(theta, l_val)] < opt_res:
270                 opt_res = results[(theta, l_val)]
271                 opt_theta = theta
272                 opt_reg = l_val
273     else:
274         opt_res = np.NINF
275         for theta, l_val in results:
276             if results[(theta, l_val)] > opt_res:
277                 opt_res = results[(theta, l_val)]
278                 opt_theta = theta
279                 opt_reg = l_val
280 
281     return opt_theta, opt_reg, opt_res
282 
283 
284 def gaussian_rbf_glm_test(x_train, x_valid, x_test, y_train, y_valid, y_test, l_val, theta, dataset):
285 
286     x_total = np.vstack([x_train, x_valid])
287     y_total = np.vstack([y_train, y_valid])
288 
289     # Create and Populate the Gram Matrix (K) at the current theta value
290     shape = (len(x_total), len(x_total))
291     K = np.empty(shape)  # Gram Matrix
292     prev_computed = {}  # Store previously computed gaussian kernel
293     for i in range(len(x_total)):
294         for j in range(len(x_total)):
295             a = x_total[i]
```

```
296             b = x_total[j]
297             if str((a, b)) not in prev_computed:
298                 prev_computed[str((a, b))] = gaussian_rbf(a, b, theta)
299                 prev_computed[str((b, a))] = prev_computed[str((a, b))]
300             # Add kernel to K (Gram) Matrix
301             K[i, j] = prev_computed[str((a, b))]
302 
303     # Only alpha changes with regularization value, thus calculate k_vector (and extend to matrix form = kM)
304     kM = np.empty((len(x_test), len(x_total)))
305     for i in range(len(x_test)):
306         # Create k vector, containing the kernel products of x_test and all x_training points
307         k = list()
308         vec = x_test[i]
309         for j in range(len(x_total)):
310             k.append(gaussian_rbf(vec, x_total[j], theta))
311         kM[i, :] = np.array(k)
312 
313     # K, kM matrix computed, compute test_error for each regularization value at current theta
314 
315     # Cholesky Factorization of K + lambda*1, here R is lower triangular
316     R = np.linalg.cholesky((K + l_val * np.eye(len(K))))
317     # Find inverse, P = inv(R) makes it quicker to find matrix inverse
318     P = np.linalg.inv(R)
319     # Estimate dual-variables alpha
320     alp = np.dot(np.dot(P.T, P), y_total)
321 
322     if dataset == 'mauna_loa' or dataset == 'rosenbrock':
323         # Compute predictions
324         predictions = np.dot(kM, alp)
325         # Compute model test_error at current regularization-theta pair
326         test_error = compute_rmse(y_test, predictions)
327 
328     else:
329         # Compute predictions
330         predictions = np.argmax(np.dot(kM, alp), axis=1)
331         y_test = np.argmax(1 * y_test, axis=1)
332         # Compute model prediction accuracy at current regularization-theta pair
333         test_error = (predictions == y_test).sum() / len(y_test)
334 
335     return test_error
336 
337 
338 def gaussian_rbf(x0, x1, theta):
339     return math.exp(-((l2_norm(x0, x1))**2)/theta)
340 
341 
342 # ---------------------- Main Block ----------------------
343 
344 if __name__ == '__main__':
345     # All Dataset Names
346     all_datasets = ['mauna_loa', 'rosenbrock', 'pumadyn32nm', 'iris', 'mnist_small']
347     regression_sets = ['mauna_loa', 'rosenbrock', 'pumadyn32nm']
348     classification_sets = ['iris', 'mnist_small']
349 
350     # ---------------------- Question 1 ----------------------
351 
352     # print('------------------ Overall Results for Question 1 -------------------')
353     # print('')
354     #
355     # x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset('mauna_loa')
356     # l_val = glm_validation(x_train, x_valid, y_train, y_valid)
357     # test_rmse = glm_test(x_train, x_valid, x_test, y_train, y_valid, y_test, l_val)
358     #
359     # print('Optimal Regularization Parameter (validation): ' + str(l_val))
360     # print('Test Root Mean-Squared Error: ' + str(test_rmse))
361     # print('')
362 
363     # ---------------------- Question 2 ----------------------
364 
365     # print('------------------ Overall Results for Question 2 -------------------')
366     # print('')
367     #
368     # x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset('mauna_loa')
369     # test_rmse = glm_kernelized(x_train, x_valid, x_test, y_train, y_valid, y_test, 14)
```

```
370     #
371     # visualize_kernel()
372     #
373     # print('Test Root Mean-Squared Error: ' + str(test_rmse))
374     # print('')
375
376     # ---------------------- Question 3 ----------------------
377
378     # print('------------------ Overall Results for Question 3 ------------------')
379     # print('')
380     #
381     # x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset('mauna_loa')
382     # theta, reg, valid_error = gaussian_rbf_glm_valid(x_train, x_valid, y_train, y_valid, 'mauna_loa')
383     # test_rmse = gaussian_rbf_glm_test(x_train, x_valid, x_test, y_train, y_valid, y_test, reg, theta, 'mauna_loa')
384     # print('--- Results for mauna_loa ---')
385     # print('Optimal Lengthscale: ' + str(theta))
386     # print('Optimal Regularizer: ' + str(reg))
387     # print('Valid RMSE: ' + str(valid_error))
388     # print('Test RMSE: ' + str(test_rmse))
389     # print('')
390
391     # x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset('rosenbrock', n_train=1000, d=2)
392     # theta, reg, valid_error = gaussian_rbf_glm_valid(x_train, x_valid, y_train, y_valid, 'rosenbrock')
393     # test_rmse = gaussian_rbf_glm_test(x_train, x_valid, x_test, y_train, y_valid, y_test, reg, theta, 'rosenbrock')
394     # print('--- Results for rosenbrock ---')
395     # print('Optimal Lengthscale: ' + str(theta))
396     # print('Optimal Regularizer: ' + str(reg))
397     # print('Valid RMSE: ' + str(valid_error))
398     # print('Test RMSE: ' + str(test_rmse))
399     # print('')
400
401     # x_train, x_valid, x_test, y_train, y_valid, y_test = load_dataset('iris')
402     # theta, reg, valid_ratio = gaussian_rbf_glm_valid(x_train, x_valid, y_train, y_valid, 'iris')
403     # test_ratio = gaussian_rbf_glm_test(x_train, x_valid, x_test, y_train, y_valid, y_test, reg, theta, 'iris')
404     # print('--- Results for iris ---')
405     # print('Optimal Lengthscale: ' + str(theta))
406     # print('Optimal Regularizer: ' + str(reg))
407     # print('Valid Accuracy Ratio: ' + str(valid_ratio))
408     # print('Test Accuracy Ratio: ' + str(test_ratio))
409     #
410     # print('')
411
412     # ---------------------- Question 4 ----------------------
413     # Done on paper
414     # ---------------------- Question 5 ----------------------
415     # Done on paper
```