Introduction to Machine Learning

Homework Solution - 3

Instructor: Prof Chinmay Hegde *TA:* Devansh Bisla, Maryam Majzoubi

Problem 1
Suppose we wish to learn a regularized least squares model:

$$L(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \langle w, x_i \rangle)^2 + \lambda R(w)$$

where $R(w)$ is a regularization function to be determined. Suggest good choices for $R(w)$ if the following criteria need to be achieved (there are no unique correct answers) and justify your choice in a sentence or two:

a. All parameters w are free to be determined.
b. w should be sparse (i.e., only a few coefficients of w are nonzero).
c. The coefficients of w should be small in magnitude on average.
d. For most indices j, w_j should be equal to w_{j-1}.
e. w should have no negative-valued coefficients.

(Solution)

a. Since we are free to choose parameters $R(w) = 0$ is one option, however, in order to have a lower variance Ridge regression is a reasonable choice here: $R(w) = ||w||_2^2$.

b. LASSO regression is a good option, i.e. $R(w) = ||w||_1$, since it promotes many coefficients to be zero, thus leading to a sparse vector.

c. Ridge regression can be used, since it shrinks the coefficients and on average it helps to reduce the coefficients to have smaller magnitude.

d. We can explicitly have this with total variation regularization function:
$R(w) = \sum_{i=1}^{d-1} |w_i - w_{i+1}|$, which penalizes if $w_i \neq w_{i+1}$.

e. Selecting $R(w) = \sum_{i=1}^{d} e^{-cw_i}$ penalizes the negative-valued coefficients. By choosing a large enough c, we will end up with no negative-valued coefficients.

Problem 2
The *Boston Housing Dataset* has been collected by the US Census Service and consists of 14 urban quality-of-life variables, with the last one being the median house price for a given town. Code for loading the dataset is provided at the end of this assignment. Implement a linear regression model with ridge regression that predicts median house prices from the other variables. Use 10-fold cross validation on 80-20 train-test splits and report the final R^2 values that you discovered. (You may want to preprocess your data to the range $[0, 1]$ in order to get meaningful results.)

HW3_Question2

March 6, 2020

[1]:
```
import pandas as pd
import numpy as np
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn import linear_model

# load dataset
X,Y = load_boston(return_X_y=True)

# normalize dataset
X = (X - np.min(X,axis=0))/(np.max(X,axis=0) - np.min(X,axis=0))

# add a column on 1's
X = np.concatenate([np.ones((X.shape[0],1)), X],axis=1)
```

[2]:
```
def ridge_regression(X,Y):
    # We will implement our own version of ridge regression. We know
    # from the derivation that,
    #
    #         W = (X'X + lam*I)^{-1} * (X'*Y)
    #
    # Author: Devansh Bisla
    #
    # Input:
    #      X: data
    #      Y: targets
    # Output:
    #      W: weights
    #
    n = X.shape[1]
    lam = 0.1
    X_T_y = X.T @ Y
    temp = X.T@X + lam*np.eye(n)

    # Tip:
    # If Ax=b then x = A^{-1}b BUT note that it is highly discouraged to
    # compute inverse of a matrix numerically since it is computationally very
```

```
    # expensive and can be inaccurate for large
    # matrices. We instead solve linear system of equations (Ax=b)
    # using some matrix factorization.
    W = np.linalg.solve(temp, X_T_y)

    return W
```

[3]:
```
rsq_hist = []
for i in range(0,10):
    # 10-fold cross-validation
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y,test_size=0.2)
    W = ridge_regression(X_train,Y_train)

    y_pred = X_test @ W
    RSS = np.mean((y_pred-Y_test)**2)/(np.std(Y_test)**2)
    RSQ = 1 - RSS
    rsq_hist.append(RSQ)

print(np.mean(rsq_hist))
```

```
0.7351295168722342
```

Problem 3
In class, we discussed the *lasso* objective, where the regularizer was chosen to be the ℓ_1-norm. Here, we will derive an analytical closed expression for a slightly simpler problem. Suppose x is a d-dimensional input and w is a d-dimensional variable. Show that the minimizer of the loss function:

$$L(w) = \min \frac{1}{2}\|x - w\|_2^2 + \lambda\|w\|_1$$

is given by:

$$w_i^* = \begin{cases} x_i - \lambda & \text{if} \quad x_i > \lambda, \\ x_i + \lambda & \text{if} \quad x_i < -\lambda, \\ 0 & \text{otherwise.} \end{cases}$$

(Solution) We can re-write the loss function as following:

$$L(w) = \min \frac{1}{2}\sum_{i=1}^{d}(x_i - w_i)^2 + \lambda|w_i| = \min \sum_{i=1}^{d} f(w_i)$$

in order to minimize the above we take the derivative and set it to zero, which is equal to the following:

$$\frac{\partial f(w_i)}{\partial w_i} = 0 \ \texttt{for} \ i = 1, \cdots, d$$

If $w_i > 0$:

$$\frac{\partial f(w_i)}{\partial w_i} = -(x_i - w_i) + \lambda = 0$$

$$w_i^* = x_i - \lambda$$

If $w_i < 0$:

$$\frac{\partial f(w_i)}{\partial w_i} = -(x_i - w_i) - \lambda = 0$$

$$w_i^* = x_i + \lambda$$

therefore,

$$w_i^* = \begin{cases} x_i - \lambda & \text{if} \quad x_i > \lambda, \\ x_i + \lambda & \text{if} \quad x_i < -\lambda, \\ 0 & \text{otherwise.} \end{cases}$$

Problem 4

In this problem, we will implement logistic regression trained with GD/SGD and validate on synthetic training data.

a. Suppose that the data dimension d equals 2. Generate two clusters of data points with 100 points each (so that the total data size is $n = 200$), by sampling from Gaussian distributions centered at $(0.5, 0.5)$ and $(-0.5, -0.5)$. Call the data points x_i, and label them as $y_i = \pm 1$ depending on which cluster they originated from. Choose the variance of the Gaussian to be small enough so that the data points are sufficiently well separated. Plot the data points on the 2D plane to confirm that this is the case.

b. (Derive your own GD routines; do *not* use sklearn functions here.) Train a logistic regression model that tries to minimize:

$$L(w) = -\sum_{i=1}^{n} y_i \log \frac{1}{1 + e^{-\langle w, x_i \rangle}} + (1 - y_i) \log \frac{e^{-\langle w, x_i \rangle}}{1 + e^{-\langle w, x_i \rangle}}$$

using Gradient Descent (GD). Plot the decay of the training loss function as a function of number of iterations.

c. Train the same logistic regression model, but this time using Stochastic Gradient Descent (SGD). Demonstrate that SGD exhibits a slower rate of convergence than GD, but is faster per-iteration, and does not suffer in terms of final quality. You may have to play around a bit with the step-size parameters as well as mini-batch sizes to get reasonable answers.

d. Overlay the original plot of data points on the 2D data plane with the two (final) models that you obtained above in parts b and c to visualize correctness of your implementation.

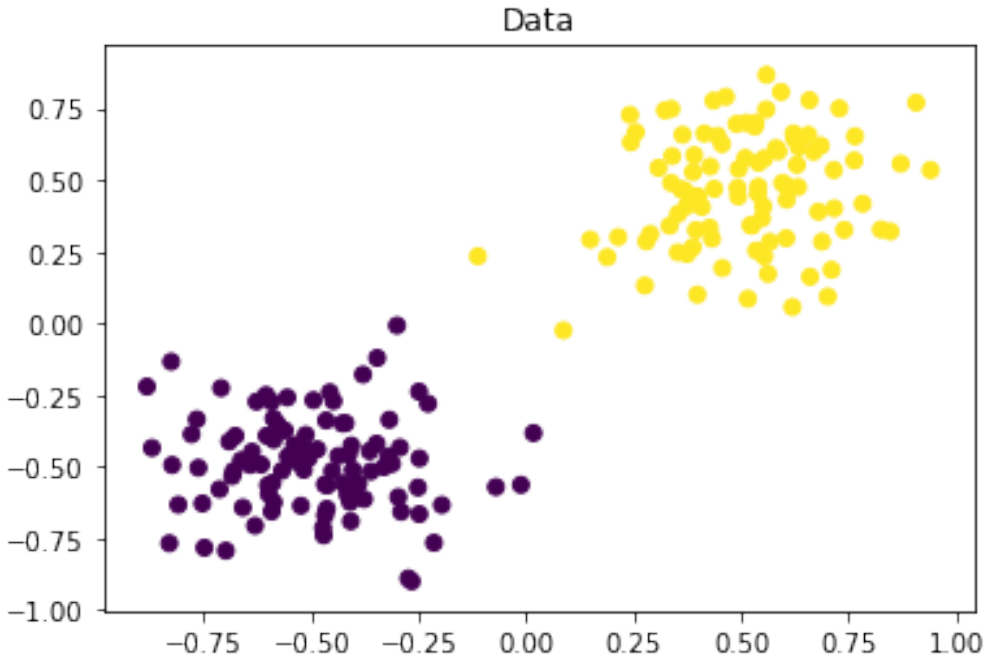
HW3_Question4

March 6, 2020

[1]:
```
%matplotlib inline
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
import random
```

[3]:
```
d = 2
means1 = np.array([0.5,0.5])
cov = 0.03*np.eye(d)
X1 = np.random.multivariate_normal(means1,cov,100)
Y1 = np.ones(len(X1))

means2 = np.array([-0.5,-0.5])
X2 = np.random.multivariate_normal(means2,cov,100)
Y2 = np.zeros(len(X1))

X, Y = np.concatenate([X1,X2]), np.concatenate([Y1,Y2])
plt.figure()
plt.scatter(X[:,0],X[:,1],c=Y)
plt.title('Data')
plt.show()

X = np.concatenate([np.ones((X.shape[0],1)), X],axis=1)
```

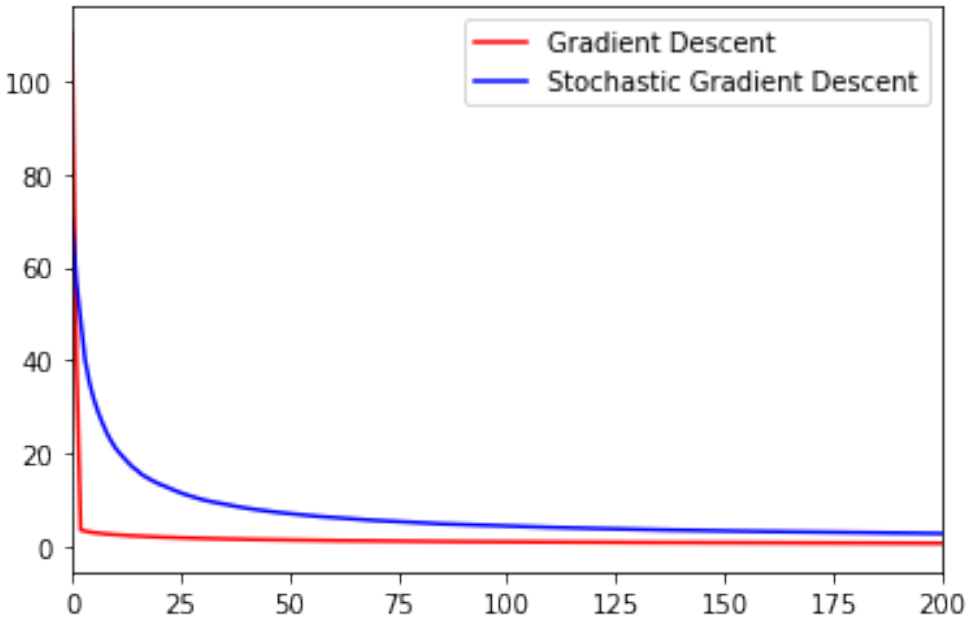


[4]:
```
lr = 0.1
N = X.shape[0]
MAX_EPOCH = 10000
w_init = np.random.rand(3)
```

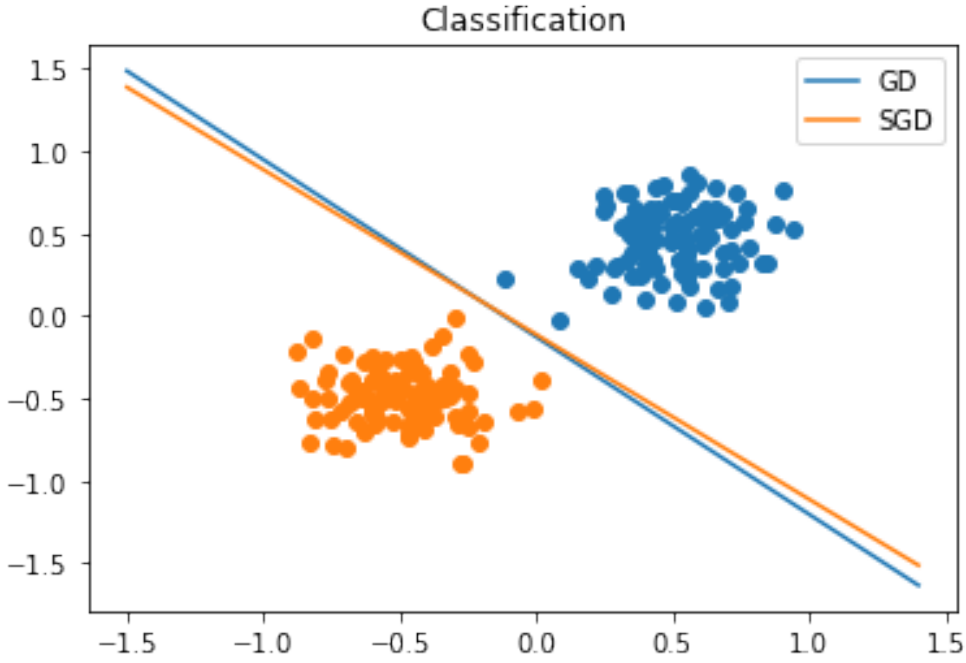
[5]:
```
# gradient descent
# intialization
W = w_init
gd_loss = []
for i in range(MAX_EPOCH):

    y_pred = X@W
    exp_ = np.exp(-1*y_pred)
    grad = -X.T@(Y - (1/(1+exp_)))

    # break condition
    if np.linalg.norm(grad) < 1e-6:
        break
    else:
        # update condition
        W = W + lr*(-grad)
        loss = -(Y.T@np.log(1/(1+exp_)) + (1-Y).T @ np.log(exp_ / (1+exp_)))
        gd_loss.append(loss)
W_gd = W
```

[6]:
```
# Stochastic gradient descent
# intialization
W = w_init
batch_size = 20
sgd_loss = []
#
# We iterate int(MAX_EPOCH/int(X.shape[0]/batch_size)) times so that
# GD and SGD both see exactly same number of weight updates
#
for i in range(int(MAX_EPOCH/int(X.shape[0]/batch_size))):
    # shuffle dataset at each epoch
    idx = np.arange(X.shape[0])
    random.shuffle(idx)
    X, Y = X[idx,:], Y[idx]

    # selecting batches
    for j in range(0,int(X.shape[0]/batch_size)):
        X_batch, Y_batch = X[batch_size*j:min(N, batch_size*j + batch_size), :
 ↪],Y[batch_size*j:min(N, batch_size*j + batch_size)]
        y_pred = X_batch@W
        exp_ = np.exp(-1*y_pred)
        grad = -X_batch.T@(Y_batch - (1/(1+exp_)))

        # break condition
        if np.linalg.norm(grad) < 1e-6:
            break
        else:
            # update condition
            W = W + lr*(-grad)

            # we compute overall loss to compare with GD
            y_pred = X@W
            exp_ = np.exp(-1*y_pred)
            loss = -(Y.T@np.log(1/(1+exp_)) + (1-Y).T @ np.log(exp_ / (1+exp_)))
            sgd_loss.append(loss)
W_sgd = W
```

[7]:
```
# Plotting both loss
plt.figure()
plt.plot(gd_loss,'r')
plt.plot(sgd_loss,'b')
plt.xlim([0,200])
plt.legend(["Gradient Descent","Stochastic Gradient Descent"])
plt.show()
```



[9]:
```
x = np.arange(-1.5,1.5,0.1)
line_GD = -(W_gd[0] + W_gd[1]*x)/W_gd[2]
line_SGD = -(W_sgd[0] + W_sgd[1]*x)/W_sgd[2]
plt.plot(x,line_GD)
plt.plot(x,line_SGD)
plt.legend(('GD','SGD'))
plt.scatter(X1[:,0],X1[:,1])
plt.scatter(X2[:,0],X2[:,1])
plt.title('Classification')
plt.show()
```


Classification
GD
SGD
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

[ ]: