Final Exam, CPSC 8420, Spring 2022

Sherrer, Will

Due 05/06/2022, Friday, 11:59PM EST

Problem 1 [15 pts]

Consider the elastic-net optimization problem:

$$\min_{\beta} \|\mathbf{y} - \mathbf{X}\beta\|^2 + \lambda[\alpha\|\beta\|_2^2 + (1-\alpha)\|\beta\|_1]. \tag{1}$$

1. Show the objective can be reformulated into a lasso problem, with a slightly different $\mathbf{X}, \mathbf{y}$.

   We can see that if alpha is 0, the ridge regression aspect of elastic net will go to 0 and we will have the Lasso formulation remaining as

   $$\min_{\beta} \|\mathbf{y} - \mathbf{X}\beta\|^2 + \lambda\|\beta\|_1 \tag{2}$$

2. If we fix $\alpha = .5$, please derive the closed solution by making use of alternating minimization that each time we fix the rest by optimizing one single element in β. You need randomly generate $\mathbf{X}, \mathbf{y}$ and initialize β_0, and show the objective decreases monotonically with updates.

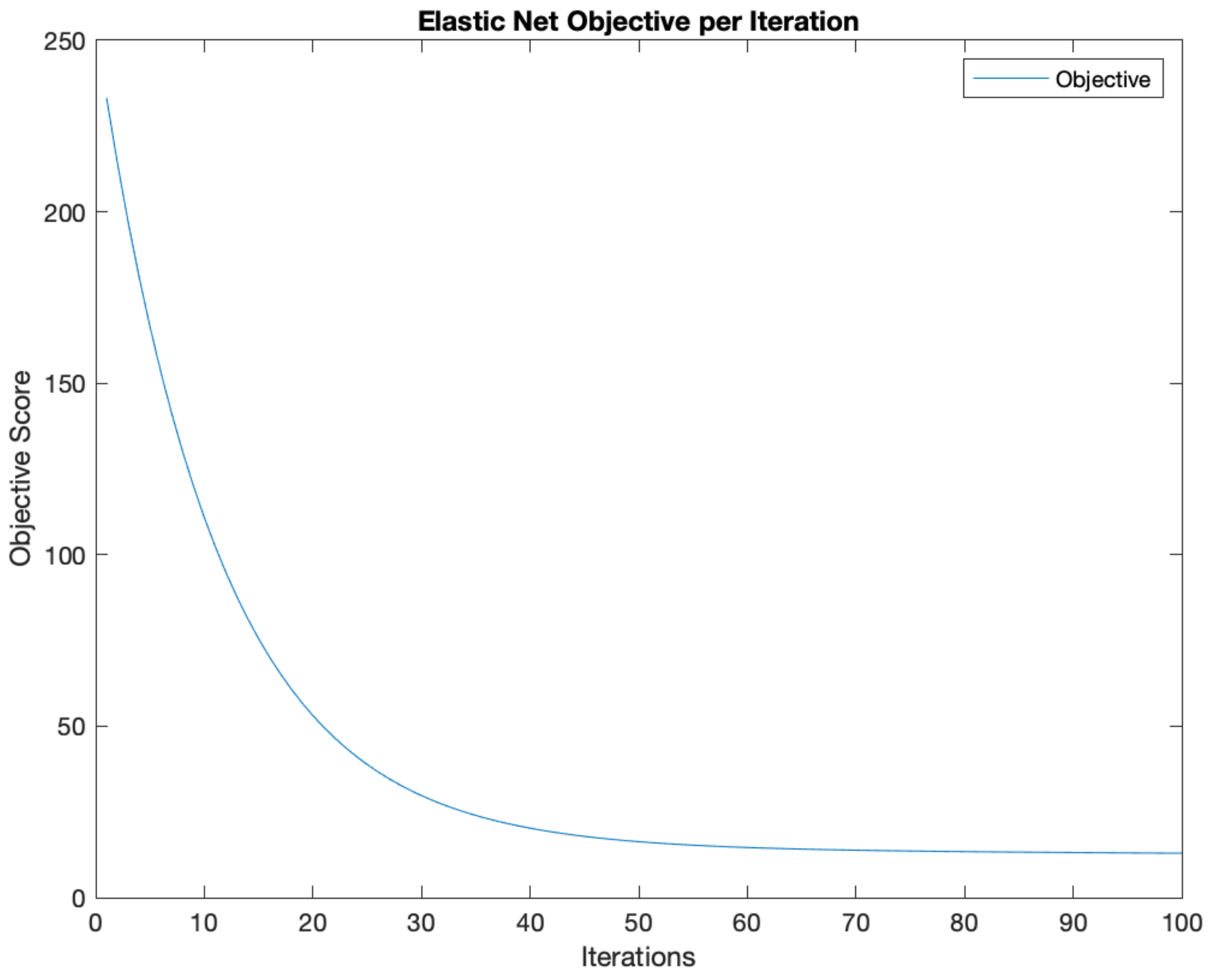
   First we can use reformulate the objective with $\alpha = 0.5$ to be:

   $$\min_{\beta} \|\mathbf{y} - \mathbf{X}\beta\|^2 + \frac{\lambda}{2}\|\beta\|_2^2 + \frac{\lambda}{2}\|\beta\|_1. \tag{3}$$

   We can take the derivative w.r.t β, however since we are also using L1 norm, we must divide into positive and negative cases where: $\hat{\beta} = -2 * X^T(y - X\beta) + \frac{\lambda}{2}\beta$

   $$\hat{\beta}^{Lasso} = \begin{cases} \hat{\beta} - \frac{\lambda}{2} & iff \hat{\beta} > \lambda \\ \hat{\beta} + \frac{\lambda}{2} & iff \hat{\beta} \leq \lambda \\ 0 & else \end{cases} \tag{4}$$

   For my implementation I am also randomly initializing β and calculating for 100 iterations with a learning rate of 0.0001 and where $\lambda = 1$.Since I am not plotting the objective with different λ since I only need to show that the objective is decreasing.



From this graph I can see that the objective of Elastic Net Objective is decreasing per iteration.

Problem 2 [15 pts]

- For PCA, the loading vectors can be directly computed from the q columns of $\mathbf{U}$ where $[\mathbf{U}, \mathbf{S}, \mathbf{U}] = svd(\mathbf{X}^T\mathbf{X})$, please show that any $[\pm\mathbf{u}_1, \pm\mathbf{u}_2, \ldots, \pm\mathbf{u}_q]$ will be equivalent to $[\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_q]$ in terms of the same variance while satisfying the orthonormality constraint.

  We know that the columns of U are orthogonal as that is in the definition of SVD. Orthogonal vectors are calculated by taking the dot product which takes the cosine of the angle between the vectors. Cosine has a symmetrical property and is an even function meaning that $cos(\theta) = cos(-\theta)$. Since the magnitude is not dependent of the signs of the vector, and the angle between the vectors is the same whether negative or positve, the dot product of both positive and negative columns will be the same which is 0. This means negative columns of U satisfies the orthonormality constraint.

  We can assume that $\phi = \sum_{i=1}^{p} \lambda_i U(:, i)$ and the constraint $||\phi||_2 = 1$ is equal to $\sum_{i=1}^{p} \lambda_i^2 = 1$. We can rewrite $X^T X = \sum_{i=1}^{p} S(i, i)U(:, i)U(:, i)^T$ and since we know the columns of U are orthogonal whether negative or positive, therefore, $||X\phi||_2^2 = \sum_{i=1}^{p} S(i, i)\lambda_i^2 \leq S(1, 1)$ Therefore the variance is the same in both cases.

- We consider the case when original dimensionality of the data is much larger than the number of samples $d \gg m$ ($\mathbf{X} \in \mathbb{R}^{d\times m}$). What's the complexity of obtaining the optimal solution of PCA via Singular Value Decomposition? Please consider a more efficient solution by considering the relationships of eigenvalues/eigenvectors between $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}\mathbf{X}^T$.

  I found that the time complexity of SVD is$O(n^3)$. In a d x m matrix when d$\gg$m, we want to use $X^T X$ so that we are performing SVD on an m x m matrix.

  For matrix X of size mxn, We can show that both $X^T X and X X^T$ are symmetric square matrices of size (n, n) and (m, m), respectively.

  $(X^T X)^T = X^T (X^T)^T = X^T X$
  $(XX^T)^T = (X^T)^T X^T = XX^T$

  It can also be shown that XX^T and $X^T X$ share the same non-zero eigenvalues. If one has more eigenvalues than the other, all of the excess eigenvalues should be zero.
  If we let $v =$ the eigenvector of $X^T X$ and λ be its eigenvalue:
  $X^T X b = \lambda v$ – if we multiply both sides by X then:
  $XX^T(Xv) = \lambda(Xv)$.
  Since λ does not $= 0$, it follows $X^T v$ does not $= 0$, therefore $X^T v$ is an eigenvector of $X^T X$ with eigenvalue of λ
  The eigenvectors to eigenvalue zero are the elements in the null space of $X : X^T v = 0$ implies $v^T X^T X v = ||Xv||^2 = 0$, which is Xv=0. You obtain the eigenvectors of $X^T X$ from $X^T v$ with v eigenvector of v of XX^T to non-zero eigenvalue. Therefore you can obtain V , whose each column is an eigenvector for XX^T, and D, a diagonal matrix holding XX^T's eigenvalues.

Problem 3 [10 pts]

Assume that in a community, there are 10% people suffer from COVID. Assume 80% of the patients come to breathing difficulty while 25% of those free from COVID also have symptoms of shortness of breath. Now please determine that if one has breathing difficulty, what's his/her probability to get COVID? (*hint*: you may consider Naive Bayes)

I want to calculate the probability of getting covid given that they have breathing difficulty. P(C|B).

I say that the probability of getting covid, P(C) is 0.1. The probability of having difficulty breathing given that they have covid is P(B|C) = 0.8.

To calculate the probability that one has breathing difficulty, I need to add the probability of breathing difficulty given covid and without covid.

$\mathrm{P(B)} = \mathrm{P(B|C)} * \mathrm{P(C)} + \mathrm{P}(\mathrm{B}|\bar{C}) * \mathrm{P}(\bar{C})$
$\mathrm{P(B)} = 0.8 * 0.1 + 0.25 * 0.9$
$\mathrm{P(B)} = .08 + .225 = .305$

$\mathrm{P(C|B)} = \frac{P(B|C)*P(C)}{P(B)} = \frac{0.8*0.1}{0.305} = \frac{0.08}{0.305} = 0.2623$ or 26.23%

Problem 4 [20 pts]

Recall the objective for RatioCut: $RatioCut(A_1, A_2, ...A_k) = \frac{1}{2}\sum_{i=1}^{k} \frac{W(A_i,\overline{A}_i)}{|A_i|}$. If we introduce indicator vector: $h_j \in \{h_1, h_2, ..h_k\}, j \in [1, k]$, for any vector $h_j \in R^n$, we define: $h_{ij} = \begin{cases} 0 & v_i \notin A_j \\ \frac{1}{\sqrt{|A_j|}} & v_i \in A_j \end{cases}$, we can prove: $h_i^T L h_i = \frac{cut(A_i,\overline{A}_i)}{|A_i|}$, and therefore:

$$RatioCut(A_1, A_2, ...A_k) = \sum_{i=1}^{k} h_i^T L h_i = \sum_{i=1}^{k} (H^T LH)_{ii} = tr(H^T LH), \tag{5}$$

thus we relax it as an optimization problem:

$$\underbrace{arg\ min}_{H}\ tr(H^T LH)\ \ s.t.\ H^T H = I. \tag{6}$$

Now let's explore Ncut, with objective: $NCut(A_1, A_2, ...A_k) = \frac{1}{2}\sum_{i=1}^{k} \frac{W(A_i,\overline{A}_i)}{vol(A_i)}$, where $vol(A) := \sum_{i\in A} d_i, d_i := \sum_{j=1}^{n} w_{ij}$.Similar to Ratiocut, we define: $h_{ij} = \begin{cases} 0 & v_i \notin A_j \\ \frac{1}{\sqrt{vol(A_j)}} & v_i \in A_j \end{cases}$. Now

1. Please show that $h_i^T L h_i = \frac{cut(A_i,\overline{A}_i)}{vol(A_i)}$.
   $h_i^T L h_i = \frac{1}{2}\sum_{m=1}\sum_{n=1} w_{mn}(h_{im} - h_{in})^2$
   $= \frac{1}{2}(\sum_{m\in A_i, n\notin A_i} w_{mn}(\frac{1}{\sqrt{vol(A_i)}} - 0)^2 + \sum_{m\notin A_i, n\in A_i} w_{mn}(0 - \frac{1}{\sqrt{vol(A_i)}})^2)$
   $= \frac{1}{2}(\sum_{m\in A_i, n\notin A_i} w_{mn}\frac{1}{vol(A_i)} + \sum_{m\notin A_i, n\in A_i} w_{mn}\frac{1}{vol(A_i)})$
   $= \frac{1}{2}(cut(A_i, \bar{A}_i)\frac{1}{vol(A_i)} + cut(\bar{A}_i, A_i)\frac{1}{vol(A_i)})$
   $= \frac{cut(A_i,\bar{A}_i)}{vol(A_i)}$

2. Show that $NCut(A_1, A_2, ...A_k) = tr(H^T LH)$.
   Since we have proved $h_i^T L h_i = \frac{cut(A_i,\overline{A}_i)}{vol(A_i)}$, therefore

$$NCut(A_1, A_2, ...A_k) = \sum_{i=1}^{k} h_i^T L h_i = \sum_{i=1}^{k} (H^T LH)_{ii} = tr(H^T LH), \tag{7}$$

3. The constraint now is: $H^T DH = I$.

4. Find the solution to $\underbrace{arg\ min}_{H}\ tr(H^T LH)\ \ s.t.\ H^T DH = I$.
   In Spectural Clustering we need to construct the Laplacian for the obtained graph and calculate the first k eigenvectors of the Laplacian. The value of k is fixed based on the number of

clusters required. We can consider the normalized Laplacian where $L = I - D^{-1}W$. Therefor for k clusters, the solution can be achieved by taking the first k eigenvectors of $I - D^{-1}W$ as the columns of H.

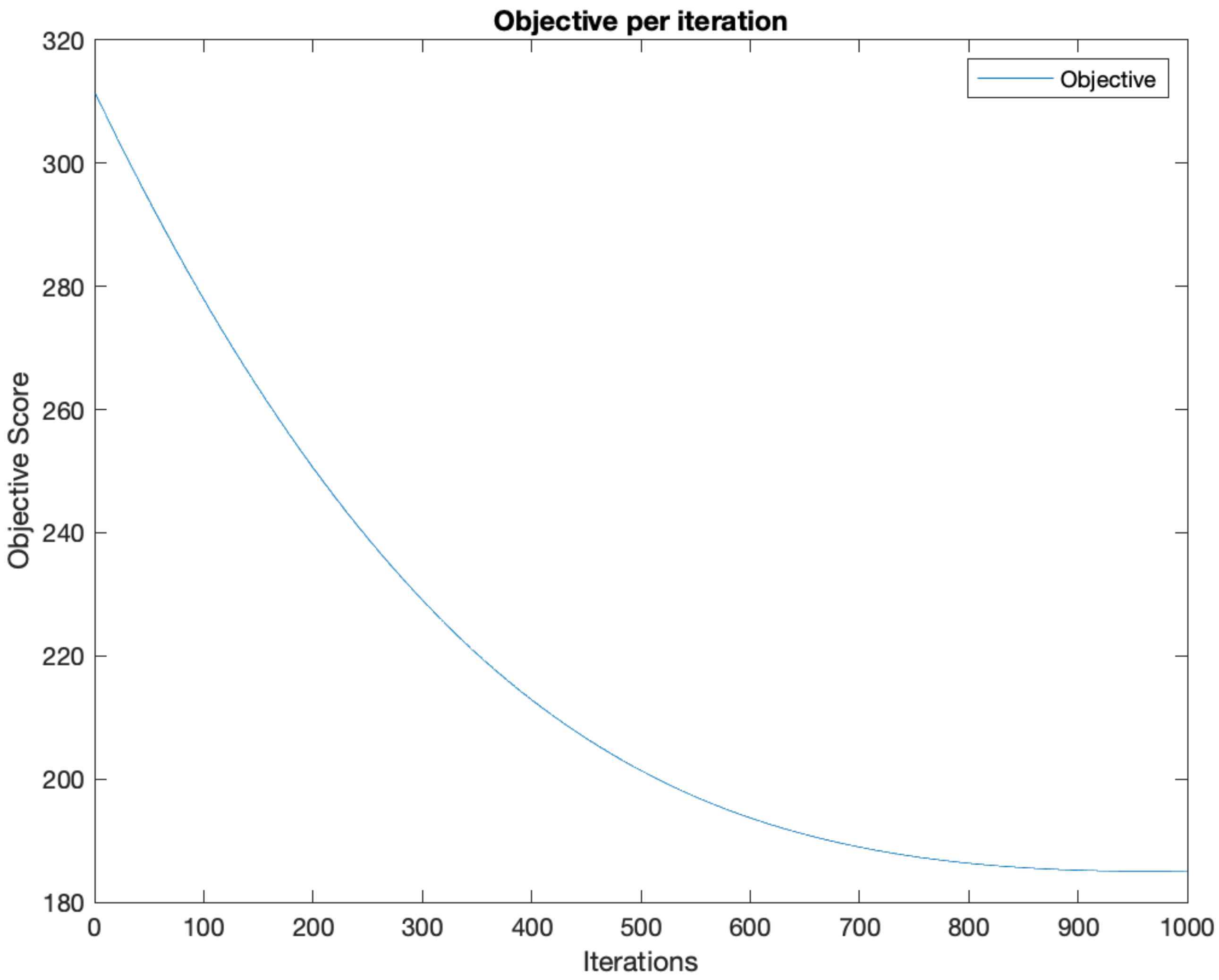
Problem 5 [10 pts]

We consider the following optimization problem ($\mathbf{Y}$ is given and generated randomly):

$$\min_{\mathbf{X}} \frac{1}{2}\|\mathbf{X}-\mathbf{Y}\|_F^2 + \|\mathbf{X}\|_* \tag{8}$$

where $\mathbf{Y}, \mathbf{X} \in \mathbb{R}^{100\times 100}$ and $\|\cdot\|_*$ denotes the nuclear norm (sum of singular values). Now please use gradient descent method to update $\mathbf{X}$. ($\frac{\partial\|\mathbf{X}\|_*}{\partial \mathbf{X}} = \mathbf{U}\mathbf{V}^T$, where $\mathbf{U}, \mathbf{V}$ is obtained from reduced SVD, namely $[\mathbf{U}, \mathbf{S}, \mathbf{V}] = svd(\mathbf{X}, 0)$). Plot the objective changes with 1000 iteration.

I first noticed that the optimization problem can be rewritten as $tr((X-Y)^T(X-Y)) + \|\mathbf{X}\|_*$ In which the gradient $w.r.t. X$ can be seen as just being the gradient $(X-Y)$ plus the nuclear norm gradient which is given as $\frac{\partial\|\mathbf{X}\|_*}{\partial \mathbf{X}} = \mathbf{U}\mathbf{V}^T$. Through a web search on Matlab, I found that calculating the nuclear norm is as follows, $nucnorm = norm(svd(X), 1)$; In my code I used a lambda of .001.



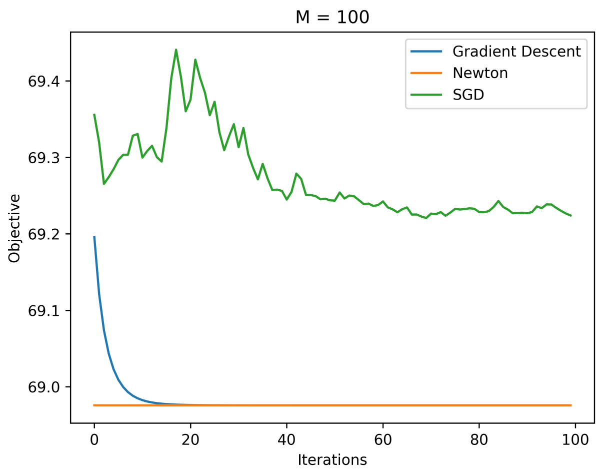
Problem 6 [20 pts]

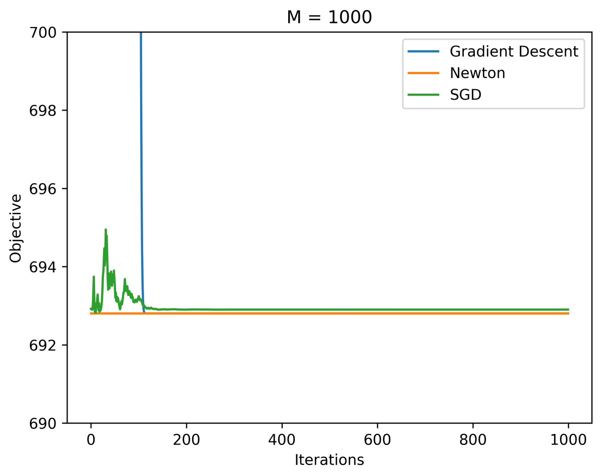
We turn to Logistic Regression:

$$\min_{\beta} \sum_{i=1}^{m} ln(1 + e^{\langle \beta, \hat{x}_i \rangle}) - y_i \langle \beta, \hat{x}_i \rangle, \tag{9}$$

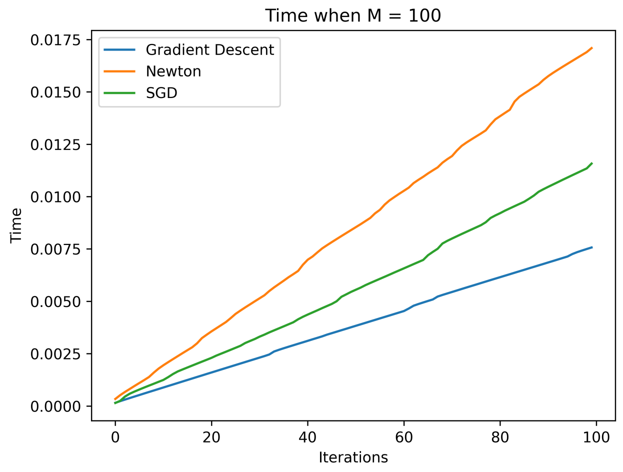
where $\beta = (w; b), \hat{x} = (x; 1)$. Assume $m = 100, x \in \mathbb{R}^{99}$. Please randomly generate x, y and find the optimal β via 1) gradient descent; 2) Newton's method and 3) stochastic gradient descent (SGD) where the batch-size is 1. (need consider choosing appropriate step-size if necessary). Change $m = 1000, x \in \mathbb{R}^{999}$, observe which algorithm will decrease the objective faster in terms of iteration (X-axis denotes number of iteration) and CPU time. [You will receive another 5 bonus points if you implement backtracking line search]

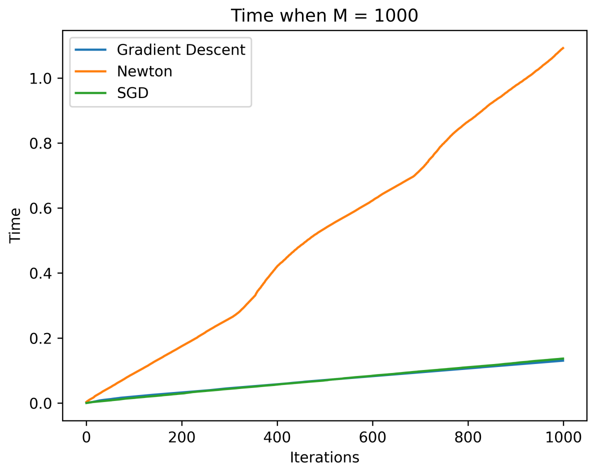
Norm of the matrix per iteration





Time per iteration





There are several observations I made from these graphs.

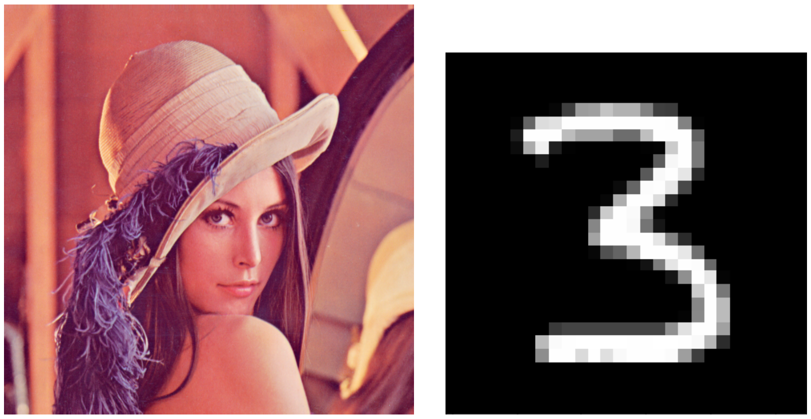
First I see that SGD does not converge, while is this unusual it is not expected to converge since the batch size is 1 and I am taking the gradient of 1 random training sample. This wouldn't matter as much however x and y are randomly generated with no specific pattern. Therefore the gradient of 1 point may be very different from the gradient of another. When M = 1000 Gradient descent appears from the top and quickly converges to 0. I found that in the beginning the Norm is in the 3000s and then it starts to converge. I believe this is because with a higher training set there is room for higher error. I also noticed that for all random data, the best β was normally not far from [0, 0],from my understanding this is because none of the data has any real world pattern since it is all randomly generated. Because of the close intial guess, Newton's method converge within < 5 iterations which does not appear on the graphs.

When M = 100 Newton's method takes the longest unsurprisingly, however SGD takes 5 milliseconds longer than Gradient Descent. I believe that in this case the extra line of code in the SGD gradient calculation creating a random number accounts for this difference as I am calling an external function. This hypothesis is further justified as SGD is narrowly faster than Gradient Descent when M = 1000. Therefore I believe when M is Larger SGD will be faster than Gradient Descent, and at that time Newton's method would outlive the entire human race with how long it would take to calculate.

My code is at the bottom of exam, I didn't include the plot code as it is just plotting the data.

Problem 7 [10 pts]

Please design an (either toy or real-world) experiment to demonstrate that PCA can be helpful for denoising.
In my experiment I took the Lenna image and an image from the MNIST dataset. I then added noise to it and performed PCA by SVD of the first 50 columns of each image. Code will be included at end of exam.


Original Image

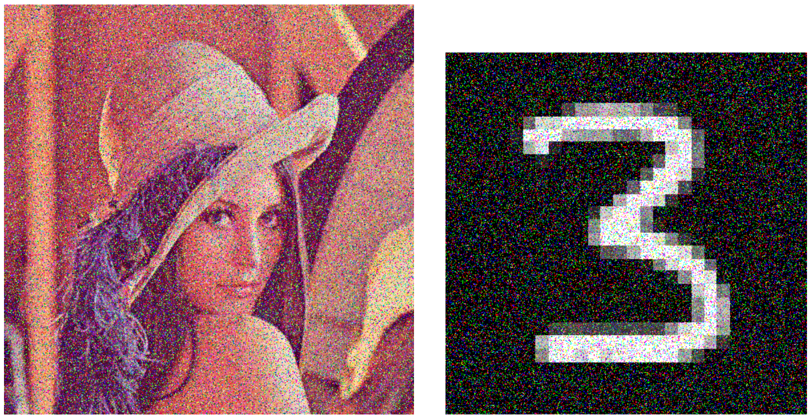


Image with noise

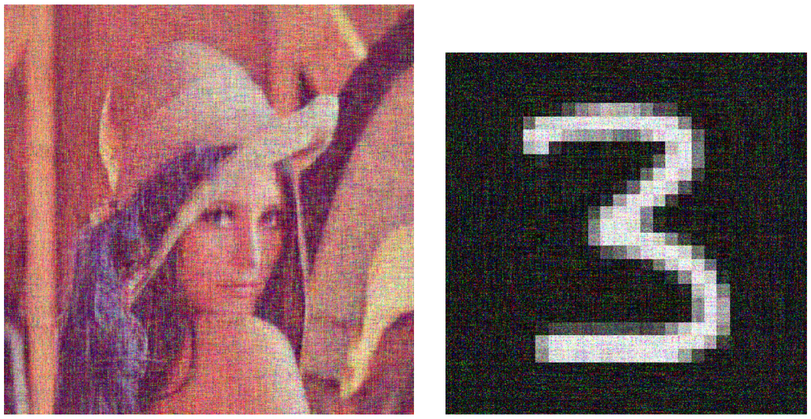


Image with reduced noise by PCA

Bonus Problem 8 [10 pts]

$$\textbf{Solve:} \quad \min \|\mathbf{x}\|_0 \quad \text{s.t.} \quad \mathbf{A}\mathbf{x} = \mathbf{y}. \tag{10}$$

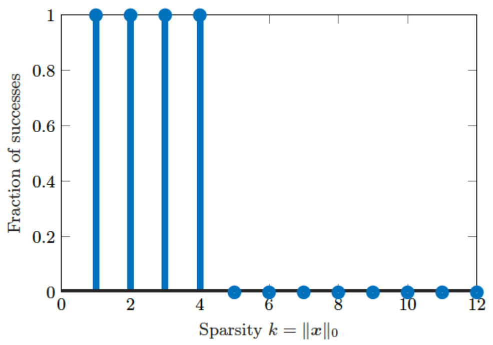
We have proved that if $\mathbf{y} = \mathbf{A}\mathbf{x}_o$ with

$$\|\mathbf{x}_o\|_0 \ \leq \ \tfrac{1}{2}\,\text{krank}(\mathbf{A}). \tag{11}$$

Then $\mathbf{x}_o$ is the unique optimal solution to the ℓ^0 minimization problem

$$\min \|\mathbf{x}\|_0 \quad \text{s.t.} \quad \mathbf{A}\mathbf{x} = \mathbf{y}. \tag{12}$$

However, when $\mathbf{A}$ is of size 5×12, the following figure illustrates the fraction of success across 100 trials. Apparently $krank(\mathbf{A}) \leq rank(\mathbf{A}) \leq 5$, therefore, when sparsity $k = 1, 2$ satisfying Eq. (11)



it has 100% recovery success rate is not surprising. However, the above experiment also shows even $k = 3, 4$ which violates Eq. (11), still it can be recovered at 100%. Please explain this phenomenon.

Proof:

$$A\hat{x} = y \rightarrow A(\hat{x} - x_0) = A\hat{x} - Ax_0 = y - y = 0 \tag{13}$$

This phenomenon can be explain because while the data theoretically cannot be recovered in the ℓ^0 norm, it can be recovered in the ℓ^1 norm. While ℓ^0 is not convex, ℓ^1 is and can be used as a convex surrogate for ℓ^0. $|x|$ is the convex envelope for $||x||_0$. We know from experimental results like the one above that ℓ^1 minimization succeeds with very high rate whenever the constants of n/m and k/m are relatively small. Here we introduce the Restricted Isometry Property (RIP) of order k with constant $\delta \in [0,\,1)$ if

$$\forall\ \mathbf{x}\ k - sparse, \quad (1-\delta)||x||_2^2 \leq ||\mathbf{A}\mathbf{x}||_2^2 \leq (1=\delta)||x||_2^2 \tag{14}$$

For ℓ^0 norm, we say that $y = Ax_o$, with $k = ||x_o||_0$. If $\delta_{2k}(A) < 1$, then x_o is the unique optimal solution to

$$min||x||_0 \quad s.t.\ Ax = y \tag{15}$$

We can prove this by contradiction, let's say there exist an $x' \neq x$ with $||x'||_o \leq k$ Then $x_o - x' \in null(A)$, and $||x_o - x'||_0 \leq 2k$. This implies that $\delta_{2k}(A) \geq 1$, contradicting our statement and therefore we can conclude the original statement is correct.

For ℓ^1 norm recovery under RIP, we say that $y = Ax_o$, with $k = ||x_o||_0$. If $\delta_{2k}(A) < \sqrt{2} - 1$, then x_o is the unique optimal solution to:

$$min||x||_1 \quad s.t.\ Ax = y \tag{16}$$

Candes and Tao proved in 2006 originally that $\delta_{2k} < \sqrt{2} - 1$

Computing the RIP constant $\delta_k(A)$ is an np-hard problem, however we know that there exists a numerical constant C > 0 such that if $A \in \mathbb{R}^{mxn}$ is a random matrix with entries independent $\mathcal{N}(0, \frac{1}{m})$ random variables, with high probability, $delta_k(A) < \delta$, provided that

$$m \geq Ck\, log(n/k)/\delta^2 \tag{17}$$

With incoherence, we need $m \geq \Omega(k^2)$. Here this result allows (k,m,n) to scale proportionally: $m \geq \Omega(k)$.

1 Code

1.1 Minmize Elastic Net Objective Problem 1

```
clc; clear;

rng('default');

X = rand(100, 9);
y = rand(100, 1);
beta = rand(9, 1);
learning_rate = 0.0001;
ob_scores = [];
t = y - X * beta;
for i=1:100
    grad = beta - 2 * X' * t + 0.5 * sign(beta);
    beta = beta - learning_rate * grad;
    t = y - X * beta;
    ob = (norm(t)^2) + (0.5 * norm(beta)^2) + (0.5 * norm(beta, 1));
    ob_scores = [ob_scores ob];
end

h = figure();
plot(ob_scores)
xlabel("Iterations")
ylabel("Objective Score")
title("Elastic Net Objective per Iteration")
legend(["Objective"])
saveas(gcf, "elastic_net_ob.png")
waitfor(h)
```

1.2 Minimie Objective Problem 5

```
clc; clear;

%want to have the same random samples to replicate results
rng('default');

X = rand(100, 100);
Y = rand(100, 100);
lambda = 0.001;
ob = objective_p5(X, Y);
ob_scores = [];
for i=1:1000
    [U,S,V] = svd(X, 0);
    gradient = (X - Y) + U*V';
    X = X - lambda * gradient;
    ob = objective_p5(X, Y);
    %add objective to array for graph
    ob_scores = [ob_scores ob];
end

h = figure();
plot(ob_scores);
ylabel("Objective Score")
xlabel("Iterations")
title("Objective per iteration")
legend(["Objective"]);
saveas(gcf, 'ob_min.png')
waitfor(h);
```

```
function [cost] = objective_p5(x, y)
    cost = 0.5 * norm(x - y, "fro") + norm(svd(x), 1);
end
```

1.3 Logistic Regression Problem 6

```
import numpy as np
import time

#objective
def f(beta, X, Y):
    return np.ravel(np.ones(len(Y))*(np.log(1+np.exp(X*beta)))-Y.T*X*
        beta)[0]

#first derivative of objective
def derivative_grad_f(beta, X, Y):
    return X.T*(1/(1+1/np.exp(X*beta))-Y)

#SGD is different in vanilla gradient descent in that it randomly
    chooses a batch of x rather than use the whole dataset.
#The final said that the batch_size = 1
def derivative_SGD_f(beta, X, Y, m):
    i = np.random.randint(0, m)
    return X[i, :].T*(1/(1+1/np.exp(X[i, :]*beta))-Y[i, :])

#second derivative of objective
def second_derivative_f(beta, X, Y):
    return X.T*(np.diag(np.ravel(np.exp(X*beta)/np.power(1+np.exp(X*
        beta),2)))*X)

if __name__ == "__main__":

    #create data arrays
    objective_gd_100 = []
    objective_new_100 = []
    objective_sgd_100 = []
    time_gd_100 = []
    time_new_100 = []
    time_sgd_100 = []

    objective_gd_1000 = []
    objective_new_1000 = []
    objective_sgd_1000 = []
    time_gd_1000 = []
    time_new_1000 = []
    time_sgd_1000 = []

    size_array = [100, 1000]
    for i in range(len(size_array)):

        m = size_array[i]
        X = np.matrix(np.random.rand(m, 2) * 10)
        Y = np.matrix(np.random.randint(0, 2, (m, 1)))
        beta = np.array([[0.], [0.]])
        learning_rate = 0.001

        t = time.time()
        counter = 0
        #gradient descent
        for i in range(m):
            counter += 1
            beta = beta - learning_rate*derivative_grad_f(beta, X, Y)
            learning_rate = learning_rate/1.02
            if m == 100:
                objective_gd_100.append(f(beta, X, Y))
                time_gd_100.append(time.time() - t)
            else:
                objective_gd_1000.append(f(beta, X, Y))
                time_gd_1000.append(time.time() - t)

        print('iter =',counter)
        print(beta)
        print('norm =',np.linalg.norm(derivative_grad_f(beta, X, Y)))
        print("\n")

        beta = np.array([[0.], [0.]])
```

```python
        counter = 0
        t = time.time()
        #newton's method
        for i in range(m):
            counter += 1
            beta = beta -  np.linalg.inv(second_derivative_f(beta, X, Y))*derivative_grad_f(beta, X, Y)
            if m == 100:
                objective_new_100.append(f(beta, X, Y))
                time_new_100.append(time.time() - t)
            else:
                objective_new_1000.append(f(beta, X, Y))
                time_new_1000.append(time.time() - t)

        print('iter =',counter)
        print(beta)
        print('norm =',np.linalg.norm(derivative_grad_f(beta, X, Y)))
        print("\n")

        counter = 0
        beta = np.array([[0.], [0.]])
        t = time.time()
        learning_rate = 0.001
        #Stochastic Gradient Descent
        for i in range(m):
            counter += 1
            beta = beta - learning_rate*derivative_SGD_f(beta, X, Y, m)
            #lower the learning rate over time to prevent jumping
            learning_rate = learning_rate/1.02
            if m == 100:
                objective_sgd_100.append(f(beta, X, Y))
                time_sgd_100.append(time.time() - t)
            else:
                objective_sgd_1000.append(f(beta, X, Y))
                time_sgd_1000.append(time.time() - t)

        print('iter =',counter)
        print(beta)
        print('norm =',np.linalg.norm(derivative_SGD_f(beta, X, Y, m)))
        print("\n")

        if m == 100:
            np.save("data/objective_gd_100.npy", np.array(objective_gd_100))
            np.save("data/objective_new_100.npy", np.array(
                objective_new_100))
            np.save("data/objective_sgd_100.npy", np.array(
                objective_sgd_100))
            np.save("data/time_gd_100.npy", np.array(time_gd_100))
            np.save("data/time_new_100.npy", np.array(time_new_100))
            np.save("data/time_sgd_100.npy", np.array(time_sgd_100))
        else:
            np.save("data/objective_gd_1000.npy", np.array(
                objective_gd_1000))
            np.save("data/objective_new_1000.npy", np.array(
                objective_new_1000))
            np.save("data/objective_sgd_1000.npy", np.array(
                objective_sgd_1000))
            np.save("data/time_gd_1000.npy", np.array(time_gd_1000))
            np.save("data/time_new_1000.npy", np.array(time_new_1000))
            np.save("data/time_sgd_1000.npy", np.array(time_sgd_1000))
```

1.4 Denoise Problem 7

```
clear;clc;
for i=1:2
    if i == 1
        A = imread("3.png");
    else
        A = imread("Lenna.png");
    end
    A = imnoise(A, 'salt & pepper', .2);
    h = figure();
    imshow(A)
    waitfor(h);
    if i == 1
        imwrite(A,"3_noise.png");
    else
        imwrite(A,"Lenna_noise.png");
    end

    k=50;
    R = A(:, :, 1);
    G = A(:, :, 2);
    B = A(:, :, 3);

    IR = im2double(R);
    IG = im2double(G);
    IB = im2double(B);

    [u1,s1,v1] = svd(IR);
    [u2,s2,v2] = svd(IG);
    [u3,s3,v3] = svd(IB);

    c1 = zeros(size(IR));
    c2 = zeros(size(IG));
    c3 = zeros(size(IB));

    c1 = u1(:, 1:k) * s1(1:k, 1:k) * v1(:, 1:k)';
    c2 = u2(:, 1:k) * s2(1:k, 1:k) * v2(:, 1:k)';
    c3 = u3(:, 1:k) * s3(1:k, 1:k) * v3(:, 1:k)';
    siz1 = size(IR(:, 1));
    siz1 = siz1(1);
    siz2 = size(IR(1, :));
    siz2 = siz2(2);
    q = zeros(siz1, siz2, 3);
    q(:, :, 1) = c1;
    q(:, :, 2) = c2;
    q(:, :, 3) = c3;
    h = figure();
    imshow(q)
    waitfor(h);
    if i == 1
        imwrite(q, "3_reduced_noise.png");
    else
        imwrite(q,"Lenna_reduced_noise.png");
    end
end
```