CSE 446 HW 2

Zachary McNulty

May 2019

1 Convexity and Norms

1.1 Problem 1.1

1. Part 1

We will begin by showing $|a+b| \leq |a| + |b|$.

Case 1: $a, b \geq 0$
Since both $a, b \geq 0$ then $|a| = a, |b| = b$ by the definition of absolute value. Furthermore, it is certainly true that $a + b \geq 0$. Thus by the definition of absolute value this implies $|a + b| = a + b$.

$$|a+b| = a + b = |a| + |b|$$

Case 2: $a, b < 0$

Since both $a, b < 0$ then by the definition of absolute value $|a| = -a, |b| = -b$. Furthermore, $a, b < 0$ implies $a + b < 0$. Thus, by the definition of the absolute value we have that $|a + b| = -(a + b)$:

$$|a+b| = |(a+b)| = -(a+b) = -a - b = |a| + |b|$$

Case 3: Without loss of generality, $a < 0, b \geq 0$ with $b \geq -a$

As $a < 0$ and $b \geq 0$ then by the definition of absolute value $|a| = -a$ and $|b| = b$. Since $b \geq -a$ then $b + a \geq 0$. Thus by the definition of absolute value $|a + b| = a + b$:

$$|a+b| = a + b = |b| - |a| \leq |b| + |a|$$

Where the final inequality holds as $|x| \geq 0$ for all x.

Case 4: Without loss of generality, $a < 0, b \geq 0$ with $b < -a$

As $a < 0$ and $b \geq 0$ then by the definition of absolute value $|a| = -a$ and $|b| = b$. Since $b < -a$ then $b + a < 0$. Thus by the definition of absolute value $|a + b| = -(a + b)$:

$$|a+b| = -(a+b) = -a - b = |a| - |b| \leq |a| + |b|$$

Where the final inequality holds as $|x| \geq 0$ for all x.

Thus, in all cases we can see that at least $|a + b| \leq |a| + |b|$.

Q.E.D.

We will now show $|ab| = |a||b|$.

Case 1: $a, b \geq 0$
This implies $|a| = a, |b| = b$ and $ab \geq 0$. Since $ab \geq 0$ we know $|ab| = ab$. Thus:

$$|ab| = ab = |a||b|$$

Case 2: $a, b < 0$
This implies $|a| = -a, |b| = -b$ and $ab \geq 0$. Since $ab \geq 0$ we know $|ab| = ab$. Thus:

$$|ab| = ab = (-a)(-b) = |a||b|$$

Case 3: Without loss of generality assume $a < 0, b \geq 0$

This implies $|a| = -a, |b| = b$ and $ab \leq 0$. Since $ab \leq 0$ we know $|ab| = -ab$. Thus:

$$|ab| = -ab = (-a)(b) = |a||b|$$

Thus, under all possible cases $|ab| = |a||b|$

(a) i. By the definition of the absolute value, $|a| \geq 0$ for all $a \in \mathbb{R}$. Since the sum of non-negative numbers is also non-negative, it holds that:

$$\sum_{i=1}^{n} |x_i| = |x_1| + |x_2| + ... + |x_n| \geq 0$$

Thus this norm is non-negative. Suppose $x \neq \vec{0}$ but $||x||_1 = 0$. Since $x \neq \vec{0}$ there exists an $x_k \neq 0$. Thus by the definition of absolute value $|x_k| > 0$. As we assumed $||x||_1 = 0$ this implies:

$$||x||_1 = \sum_{i=1}^{n} |x_i| = \sum_{i \neq k} (|x_i|) + |x_k| = 0$$

$$\sum_{i \neq k} |x_i| = -|x_k| < 0$$

The left hand side represents the norm of some vector in $\mathbb{R}^{n-1}$ and we showed this is non-negative, but the right hand side of this equation is strictly less than zero. This is a contradiction, and thus there cannot exist an $x \neq \vec{0}$ such that $||x||_1 = 0$. Furthermore, if $x = \vec{0}$ then:

$$||x||_1 = ||\vec{0}||_1 = \sum_{i=1}^{n} |0| = 0 + 0 + ... + 0 = 0$$

Thus $||x||_1 = 0$ iff $x = \vec{0}$.

ii. Next we will show absolute scalability. As we showed earlier, $|ab| = |a||b|$ for all $a, b \in \mathbb{R}$. Thus:

$$||ax||_1 = \left|\left| \begin{bmatrix} ax_1 \\ ax_2 \\ ... \\ ax_n \end{bmatrix} \right|\right|_1 = \sum_{i=1}^{n} |ax_i| = \sum_{i=1}^{n} |a||x_i| = |a| \sum_{i=1}^{n} |x_i| = |a|||x||_1$$

Thus the absolute scalability condition of the norm is satisfied.

iii. Lastly, we will show the triangle inequality is satisfied. We showed above that for any $a, b \in \mathbb{R}^n$ it holds that $|a + b| \leq |a| + |b|$. Thus for any $x, y \in \mathbb{R}^n$:

$$||x+y||_1 = \left|\left| \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ ... \\ x_n+y_n \end{bmatrix} \right|\right|_1 = \sum_{i=1}^{n} |x_i+y_i| \leq \sum_{i=1}^{n} (|x_i|+|y_i|) = \sum_{i=1}^{n} |x_i| + \sum_{i=1}^{n} |y_i| = ||x||_1 + ||y||_1$$

Where the inequality step comes by applying $|a+b| \leq |a|+|b|$ for every term in the sum individually. Thus, $||x+y||_1 \leq ||x||_1 + ||y||_1$ and the triangle inequality holds.

Thus as all three properties hold $||x||_1$ is a norm.

(b) Consider the case when $x = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Thus $x+y = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. Then it holds that:

$$||x|| = \left(\sum_{i=1}^{2} |x_i|^{1/2} \right)^2 = (1^{1/2} + 0^{1/2})^2 = 1^2 = 1$$

$$||y|| = \left(\sum_{i=1}^{2} |y_i|^{1/2} \right)^2 = (0^{1/2} + 1^{1/2})^2 = 1^2 = 1$$

$$||x+y|| = \left(\sum_{i=1}^{2} |(x+y)_i|^{1/2} \right)^2 = (1^{1/2} + 1^{1/2})^2 = 2^2 = 4$$

Thus $||x+y|| > ||x|| + ||y||$ and therefore the triangle inequality does not hold. Thus, this is not a valid norm.

1.2 Problem 1.2

Firstly, note that for any $a, b \in \mathbb{R}$ such that $a, b \geq 0$ it holds that $a^2 \geq b^2$ implies $a \geq b$. We can show this shortly by contraposition. Suppose $a < b$. Then as $a, b \geq 0$:

$$a < b \rightarrow a^2 < ab$$
$$a < b \rightarrow ab < b^2$$

Thus, together these show $a^2 < ab < b^2 \rightarrow a^2 < b^2$. Thus, by contraposition $a^2 \geq b^2$ implies $a \geq b$. This is an important fact because we know for any norm $||x|| \geq 0$.

$$||x||_2^2 = \sum_{i=1}^{n} |x_i|^2 = |x_1|^2 + ... + |x_n|^2 \leq (|x_1| + |x_2| + ... + |x_n|)^2 = ||x||_1^2$$

We can see the inequality holds by distributing $(|x_1| + ... + |x_n|)^2$. This will generate all the $|x_i|^2$ terms plus many other cross terms $C|x_i||x_j|$ for some non-negative constant C (as all the coefficients in the quadratic are non-negative). Since $|x_i| \geq 0$ for all i, these cross terms can only increase (or not change) the value of the sum, hence the greater than or equal to relationship. As we showed above $||x||_2^2 \leq ||x||_1^2$ implies $||x||_2 \leq ||x||_1$.

Similarly, suppose $\max_{i=1,...,n} |x_i| = |x_k|$:

$$||x||_\infty^2 = |x_k|^2 \leq |x_1|^2 + |x_2|^2 + ... + |x_k|^2 + ... + |x_n|^2 = ||x||_2^2$$

Where the inequality above holds as $|a|^2$ is strictly non-negative for any $a \in \mathbb{R}$. As we showed above, $||x||_\infty^2 \leq ||x||_2^2$ implies that $||x||_\infty \leq ||x||_2$. Combining our findings we see that:

$$||x||_\infty \leq ||x||_2 \leq ||x||_1$$

1.3 Problem 1.3

1. Set I is not convex. We can see that the line between points b and c is not entirely contained within the set. Thus there exists $\lambda \in [0, 1]$ such that $\lambda b + (1 - \lambda)c \notin A$
2. Set II is convex. Between any two points in the set we can see the line between them remains in the set.
3. Set III is not convex. We can see that the line between points d and a is not entirely contained within the set. Thus there exists $\lambda \in [0, 1]$ such that $\lambda d + (1 - \lambda)a \notin A$

1.4 Problem 1.4

1. (a) Function I on $[a, c]$ is convex as the line between any two points $(x, f(x))$ and $(y, f(y))$ on the graph with $x, y \in [a, c]$ is never below the function value on $[x, y]$. Thus the condition $f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$ is satisfied.

   (b) Function II on $[a, c]$ is not convex. Simply choose $x = a$ and $y = b$. Then we can see that the line between $(a, f(a))$ and $(b, f(b))$ is below the function within (a, b). Thus there exists a $\lambda \in [0, 1]$ such that for some $x, y \in A = [a, c]$, i.e. $x = a, y = b$, such that $f(\lambda x + (1 - \lambda)y) > \lambda f(x) + (1 - \lambda)f(y)$.

   (c) Function III on $[a, d]$ is not convex. Simply choose $x = a$ and $y = c$ and we can see that the line between $(a, f(a))$ and $(c, f(c))$ is below the function value within (a, c). Thus there exists a $\lambda \in [0, 1]$ such that for some $x, y \in A = [a, d]$, i.e. $x = a, y = c$, such that $f(\lambda x+(1-\lambda)y) > \lambda f(x)+(1-\lambda)f(y)$.

   (d) Function III on $[c, d]$ is convex as the line between any two points $(x, f(x))$ and $(y, f(y))$ on the graph with $x, y \in [c, d]$ is never below the function value on $[x, y]$. Thus the condition $f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$ is satisfied.

1.5 Problem 1.5

1. (a) To show that $||x||$ is convex for any norm, we need to show that $f(\lambda x+(1-\lambda)y) \leq \lambda f(x)+(1-\lambda)f(y)$ for any $x, y \in \mathbb{R}^n$, the set that the norm acts on, and for any $\lambda \in [0,1]$.

$$\begin{aligned} & f(\lambda x + (1-\lambda)y) \\ & = ||\lambda x + (1-\lambda)y|| \\ & \leq ||\lambda x|| + ||(1-\lambda)y) \end{aligned}$$

Where the inequality holds by the triangle inequality that by definition all norms must satisfy. Furthermore, by the property of absolute scalability that norms must satisfy:

$$= |\lambda|||x|| + |1-\lambda|||y||$$

As $\lambda \in [0,1]$ we know $\lambda, (1-\lambda) \geq 0$. Thus:

$$= \lambda||x|| + (1-\lambda)||y||$$

$$= \lambda f(x) + (1-\lambda)f(y)$$

Thus the norm satisfies $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$ and thus the norm is convex.

(b) Suppose the set $S = \{x \in \mathbb{R}^n : ||x|| \leq 1\}$ is not convex. This implies there exists $x, y \in S$ and a $\lambda \in [0,1]$ such $\lambda x + (1-\lambda)y \notin S$. If $\lambda x + (1-\lambda)y$ is not in S then we know $||\lambda x + (1-\lambda)y|| > 1$.

$$\begin{aligned} & ||\lambda x + (1-\lambda)y|| \\ & \leq ||\lambda x|| + ||(1-\lambda)y|| \end{aligned}$$

Where the above holds by the triangle inequality that all norms satisfy. Furthermore, using the property of absolute scalability that all norms satisfy we see:

$$= |\lambda|||x|| + |1-\lambda|||y||$$

Since $\lambda \in [0,1]$, both $\lambda, (1-\lambda) \geq 0$ and thus:

$$= \lambda||x|| + (1-\lambda)||y||$$

Note that $x, y \in S$ and thus by the definition of S we see $||x||, ||y|| \leq 1$. Thus:

$$\begin{aligned} & \leq \lambda + (1-\lambda) \\ & = 1 \end{aligned}$$

But this is a contradiction to the fact that $||\lambda x+(1-\lambda)y|| > 1$. Thus we have reached a contradiction and our assumption that S was not convex must have been incorrect. Thus, S is convex.

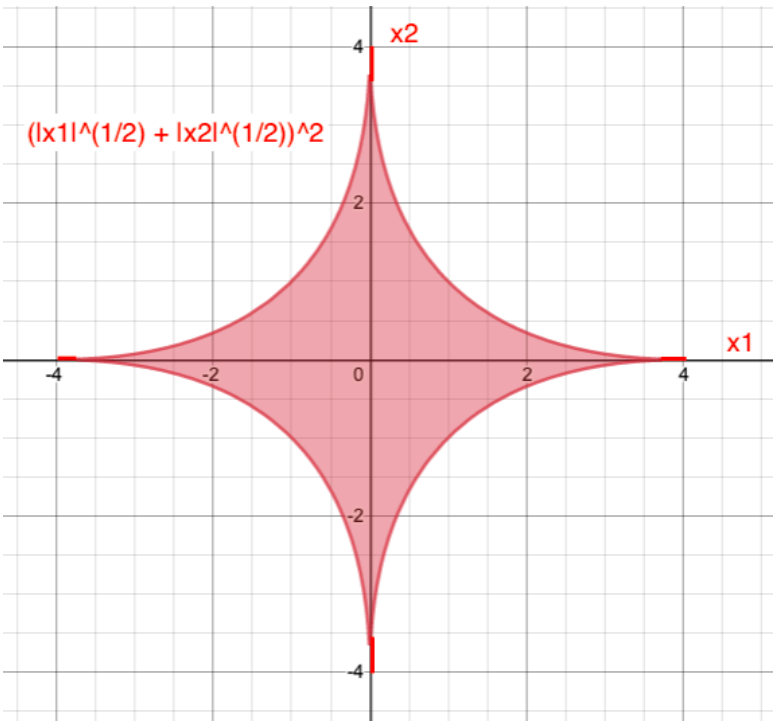

Figure 1: The set $\{(x_1, x_2) : g(x_1, x_2) \leq 4\}$ with $g(x_1, x_2) = (|x_1|^{1/2} + |x_2|^{1/2})^2$

(c) I drew the above graph using Desmos by plotting $(|x_1|^{1/2} + |x_2|^{1/2})^2 \leq 4$. As expected, the boundary of the set runs through the points $(0, 4), (4, 0), (-4, 0), (0, -4)$. Clearly this set is not convex due to the bend along the edges of the graph. In fact, we can see the line between $(0, 4)$ and $(4, 0)$ is completely outside the set so the set cannot be convex.

1.6 Problem 1.6

1. (a) Firstly, we will show if $f(x)$ and $g(x)$ are convex then their sum $h(x) = f(x) + g(x)$ is also convex. Since f, g are convex we know:

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$$

$$g(\lambda x + (1-\lambda)y) \leq \lambda g(x) + (1-\lambda)g(y)$$

This implies that:

$$\begin{aligned} h(\lambda x + (1-\lambda)y) &= f(\lambda x + (1-\lambda)y) + g(\lambda x + (1-\lambda)y) \\ &\leq \lambda f(x) + (1-\lambda)f(y) + \lambda g(x) + (1-\lambda)g(y) \\ &= \lambda(f(x) + g(x)) + (1-\lambda)(f(y) + g(y)) \\ &= \lambda h(x) + (1-\lambda)h(y) \end{aligned}$$

$$h(\lambda x + (1-\lambda)y) \leq \lambda h(x) + (1-\lambda)h(y)$$

Thus, $h(x) = f(x) + g(x)$ is convex. Next, we will show that if $f(x)$ is convex then for all $c > 0$ then $cf(x)$ is convex. Since f is convex and $c > 0$:

$$\begin{aligned} & f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \\ \rightarrow\ & cf(\lambda x + (1-\lambda)y) \leq c\left(\lambda f(x) + (1-\lambda)f(y)\right) \\ & = \lambda cf(x) + (1-\lambda)cf(y) \end{aligned}$$

Thus $cf(x)$ is convex if $c > 0$. Therefore, as we know the norm $||w||$ is convex (showed in problem 5a) then so is $\lambda||w||$ for $\lambda > 0$. Furthermore, as we showed the sum of convex functions is convex, and $\ell_i(w)$ is convex for all i, then $\ell_1(w) + \ell_2(w)$ is convex. Then this implies $(\ell_1(w) + \ell_2(w)) + \ell_3(w)$ is convex. Iteratively applying this logic, we see that $\sum_{i=1}^{n} \ell_i(w)$ is convex. Lastly, as $\lambda||w||$ is convex and $\sum_{i=1}^{n} \ell_i(w)$ is convex then:

$$\sum_{i=1}^{n} \ell_i(w) + \lambda||w||$$

is convex.

(b) Convexity guarantees that all local minima are global minima so we do not have to worry whether or not we are truly at a minimal state if we reach a local minima via whatever algorithm we use (i.e. gradient descent).

2 Programming: Lasso

2.1 Problem 7

The code for the following graphs (all those for problem 7) can be found at the end of this section. I have commented it so that it is clear which sections use each code. Both sections use the functions defined at the beginning of the document. All functions have their own comments.

I was able to speed up my code by rewriting c_k using matrix/vector operations. The math/justification for that is below. Let $x_{i\bullet}$ and $x_{\bullet k}$ stand for the i^{th} row and k^{th} column respectively of our data matrix X:

$$
\begin{aligned}
c_k &= 2\sum_{i=1}^{n} x_{ik}\left(y_i - \left(b + \sum_{j\neq k} w_j x_{ij}\right)\right) \\
&= 2\sum_{i=1}^{n} x_{ik}\left(y_i - \left(b + \sum_{j=1}^{d} w_j x_{ij} - w_k x_{ik}\right)\right) \\
&= 2\sum_{i=1}^{n} x_{ik}\left(y_i - \left(b + w^T x_{i\bullet} - w_k x_{ik}\right)\right) \\
&= 2\left(x_{\bullet k}^T \begin{bmatrix} y_1 - (b + w^T x_{1\bullet} - w_k x_{1k}) \\ \dots \\ y_n - (b + w^T x_{n\bullet} - w_k x_{nk}) \end{bmatrix}\right) \\
&= 2\left(x_{\bullet k}^T \left(y - (b + w^T X^T - w^T x_{\bullet k})\right)\right) \\
&= 2x_{\bullet k}^T \left(y - b - w^T X^T + w^T x_{\bullet k}\right)
\end{aligned}
$$

7a)

As expected, when we begin coordinate descent using λ_{max} as described in the specification (far right of the graph below) we have a weights vector completely filled with zeros. At this point, there is too much regularization and this outweighs the negatives associated to a high squared error. As we relax λ, we start getting more nonzero coefficients. However, relax it too far and we start running into issues with overfitting: all the weights begin to be nonzero, as we see towards the left of the graph below. Thus, higher values of λ are encouraging zero entries in the weight vector. Intuitively, we might think the irrelevant variables are the ones that get set to zero first as doing so might have a lower effect on the squared error than zeroing out a relevant predictor variable. However, with noise this might not always be the case. By pure chance, certain features may appear predictive in the presence of noise.

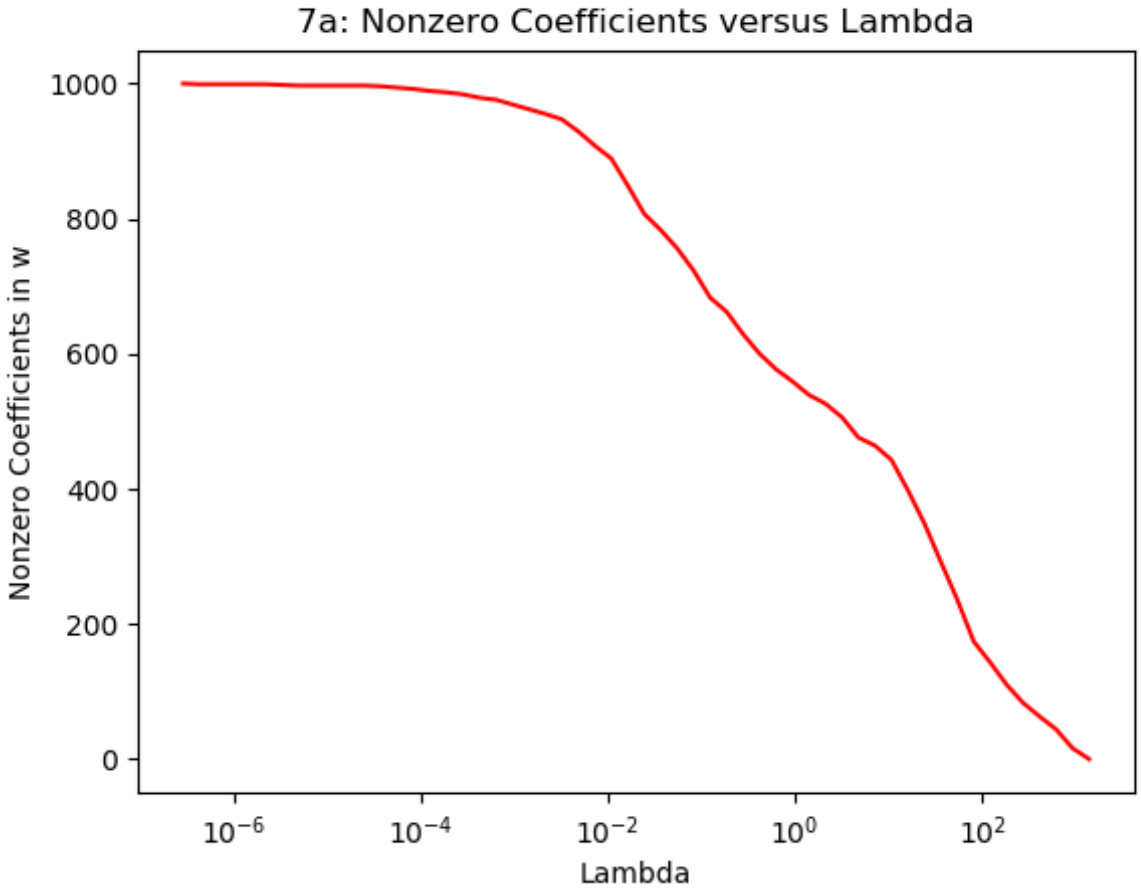

Figure 2: Problem 7a

7b)

Here we have the False discovery rate (number incorrect nonzeros / total nonzeros in w) plotted against the true positive rate (correct nonzeros / total true nonzeros). As we saw in the plot for 7a, the choice of λ plays a key roll in determining the number of nonzero entries in w. If λ is really high, we will get all zeros which will give $FDR = TPR = 0$. If our λ is really small, we will get a w that is entirely nonzero and thus $TPR = 1, FDR = \frac{d-k}{d}$. All the points in-between represent intermediate λ values. Ideally, we would like to maximize our TPR while minimizing FDR, but we can see there seems to be this clear trade-off between the two. Decreasing λ will likely increase the number of nonzeros we have, but there is no guarantee these nonzeros capture meaningful information. Conversely, increasing λ may generate more zero coefficients, but it is possible these are coefficients that are useful in predicting the response. In a real-world setting, we would have to make decisions about how important each true positive is and how costly each false discovery is to decide on this intermediate λ value.

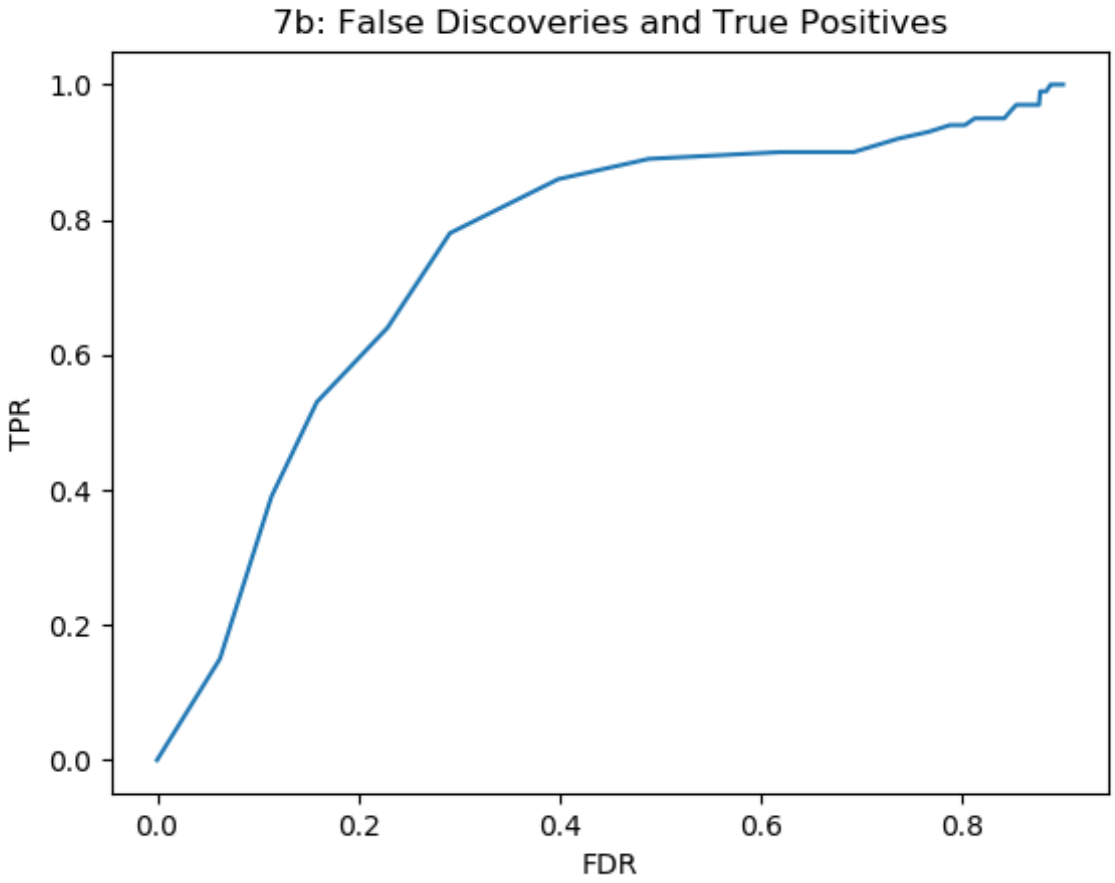

Figure 3: Problem 7a: Relationship between False Discovery Rate (FDR) and True Positive Rate (TPR). We can see there appears to be a trade-off between the two.

2.2 Problem 7 Code

```
'''
lasso.py

implements the coordinate descent algorithm for LASSO regression.
LASSO promotes sparsity by regularizing the Least Squared Error
with an L1 norm.

X --> rows are data measurements, columns are specific features
y = vector of response variables

'''

import numpy as np
import matplotlib.pyplot as plt
import time


def generate_synthetic_data(n, d, k, variance, seed=123):
    '''
    Generates synthetic data following to procedure in the homework specification.
    n = number of data samples
    d = number of features per data sample
    k = number of relevant features (i.e. whose coefficient in w we expect to be non-negative)
    variance = variance for the zero-mean gaussian noise added on top of the true system
    seed = RNG seed for reproducibility

    '''
    if k > d:
        raise ValueError("The number of relevant features cannot be more than the number of
                                                    features")
    elif n < 1 or d < 1 or k < 0 or variance < 0:
        raise ValueError("Parameters n,d >= 1 and k,variance >= 0 must be true")

    w_true = np.zeros((d ,1))

    # j+1 is to account for zero-based indexing in python
    for j in range(k): w_true[j] = (j+1) / k

    np.random.seed(seed)
    X = np.random.normal(size = (n,d))
    errors = np.random.normal(scale = np.sqrt(variance), size=(n,))

    y = np.reshape(np.dot(w_true.T, X.T) + errors.T, (n,))
    return (X, y)



def min_null_lambda(X, y):
    '''
    Returns the smallest lambda value that generates a null solution
    (i.e. w is entirely zeros). Start with lambda at this value and
    decrease over time
    Assuming y is a column vector of responses and X is a matrix with each column a feature
    and each row a measurement (i.e. the data matrix)
    '''
    return 2*np.max(np.abs(np.dot(y.T - np.mean(y), X)))



def lasso_coordinate_descent(X, y, lam, w_init = None, delta = 10e-4):
    '''
    Runs the coordinate descent algorithm on the LASSO regression problem.

    X = data matrix with each measurement stored as a row (columns are features) n x features
    y = response variable : n x 1
    lam = lambda value used for regularization
    delta = stopping condition; if no element in w changes by more than delta in a single
                                                    iteration, stop.

    '''

    n = X.shape[0]
    d = X.shape[1]
```

```
    prev_w = np.ones((d,1))
    if w_init is None:
        w = np.zeros((d,))
    else:
        w = w_init

    c = np.zeros((d,))

    # a_k is just summing over all the rows of the squared entries. Thus, we can calculate it
    # quickly by squaring every entry in X (elementwise) then summing along the rows
    a = 2*np.sum(np.square(X), axis=0)

    # while we still have elements in w that changed by more than delta in last iteration
    while np.max(np.abs(w - prev_w)) > delta:

        prev_w = np.copy(w)

        wTXT = np.dot(w.T, X.T)

        b = 1/n * np.sum(y - wTXT)

        for k in range(d):

            # NOTE: Can use updated w_k values from SAME ROUND OF ITERATION piazza @199
            #       For this reason, we have to recalculate wTXT at every iteration
            c[k] = 2*np.dot(X[:, k], y - (b + np.dot(w.T, X.T) - w[k]*X[:, k]))

            if c[k] < -1*lam:
                w[k] = (c[k] + lam) / a[k]
            elif c[k] > lam:
                w[k] = (c[k] - lam) / a[k]
            else:
                w[k] = 0

    return (w, b)


# =====================================================================================

# problem 7a)


n = 500
d = 1000
k = 100
variance = 1
(X, y) = generate_synthetic_data(n, d, k, variance, seed=3853)

lam_max = min_null_lambda(X, y)
lam_ratio = 1.5 # ratio to decrease lambda by during each iteration
delta = 10e-4 # threshold to stop iteration (search for better w)

current_lam = lam_max
lam_vals = [lam_max]
prev_w = None

W = np.zeros((d, 1)) # we will store the resulting w for each lambda as a column in this
                                                    matrix
while np.count_nonzero(W[:, -1]) != d:
    current_lam = current_lam / lam_ratio
    lam_vals.append(current_lam)

    print("Running using lambda = ", current_lam)

    # Initialize current round of coordinate descent to w found in last round
    (w_new, b) = lasso_coordinate_descent(X, y, w_init=prev_w, lam=current_lam, delta=delta)
    W = np.concatenate((W, np.expand_dims(w_new, axis=1)), axis=1)
    prev_w = np.copy(w_new)

plt.figure(1)
plt.semilogx(lam_vals, np.count_nonzero(W, axis=0), 'r-')
plt.xlabel('Lambda')
plt.ylabel('Nonzero Coefficients in w')
plt.title('7a: Nonzero Coefficients versus Lambda')
plt.show()
```

```
# Problem 7b)

# Based on the definition of w_true, only the first k entries in w are truly nonzero. To
                                                    calculate the
# FDR rate all we have to do is count the nonzero entries in the other d-k slots as incorrect
# Here we skip the first column of W as it corresponds to the w found using lambda_max, which
                                                    generates
# a w with all zeros. To avoid division by zero, we define FDR = 0 at this point
FDR = np.append([0], np.count_nonzero(W[k:, 1:], axis=0) / np.count_nonzero(W[:,1:], axis=0))

# Based on the definition of w_true only the first k entries in w are truly nonzero.
TPR = np.count_nonzero(W[:k, :], axis=0) / k

plt.figure(2)
plt.plot(FDR, TPR)
plt.title('7b: False Discoveries and True Positives')
plt.xlabel('FDR')
plt.ylabel('TPR')
plt.show()
```

3 Problem 8

The code for the following graphs (all those for problem 7) can be found at the end of this section. I have commented it so that it is clear which sections use each code. Both sections use the functions defined at the beginning of the code file. All functions have their own comments.

8a)

As was the case in problem 7, we see a clear relationship between λ and the number of nonzero coefficients in w. Increasing λ encourages sparsity, decreasing the number of nonzero coefficients. Decreasing λ leads towards a possibly overfitted state

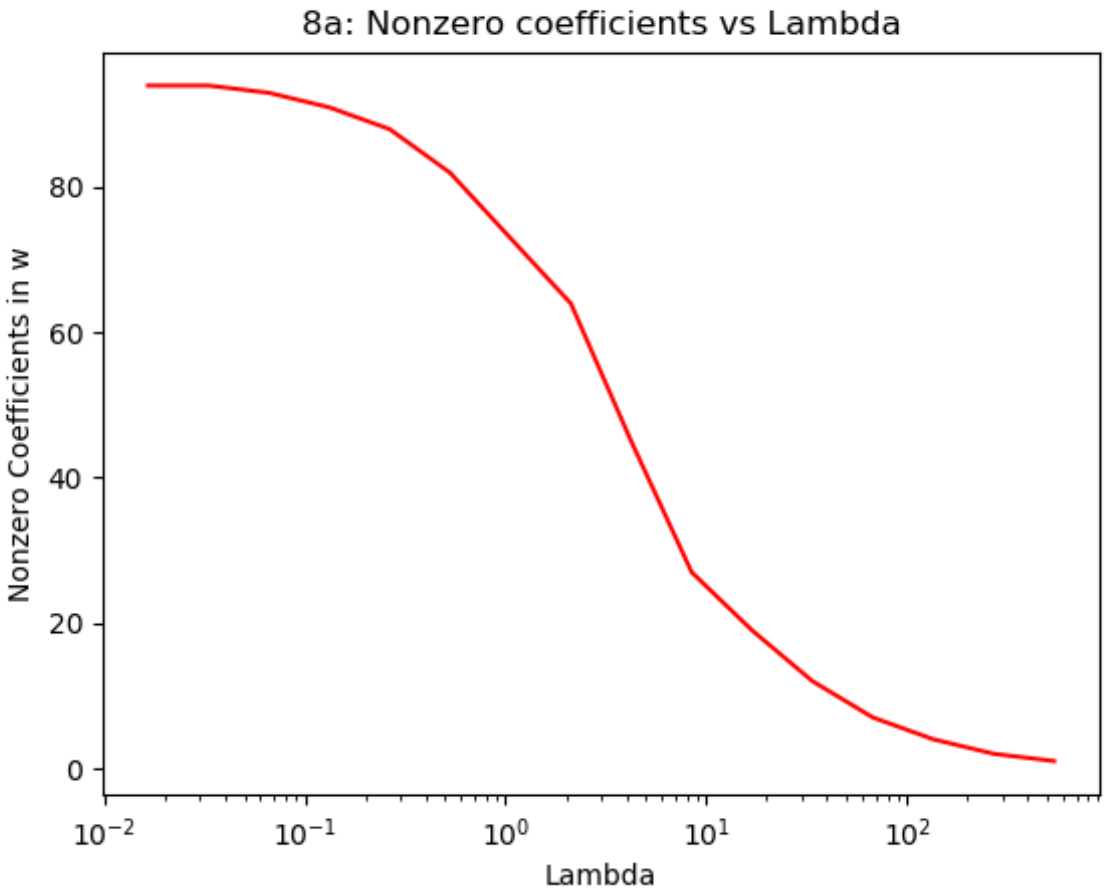

Figure 4: Problem 8a) Nonzero coefficients vs the amount of regularization. Here, we see regularization promotes sparsity in the coefficients.

8b)

As λ gets large, the coefficients are all pushed towards zero.

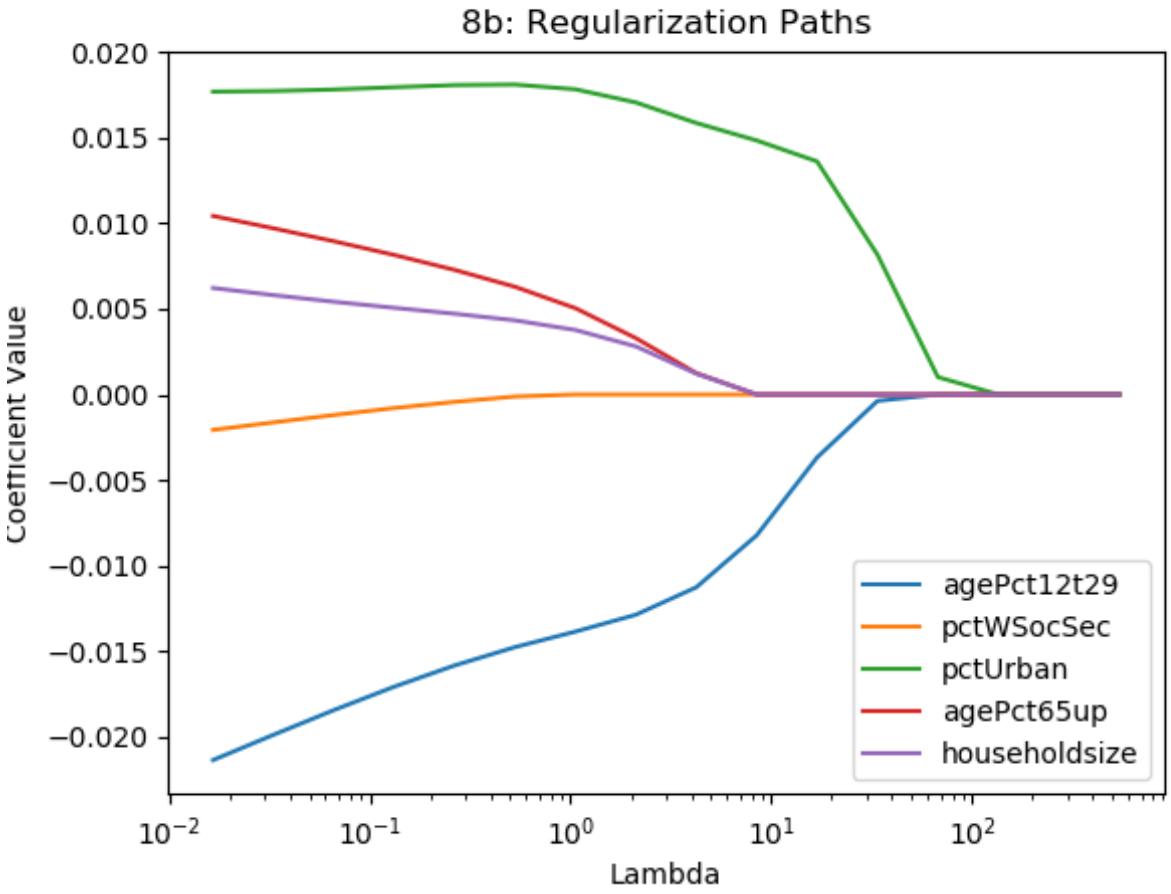

Figure 5: Problem 8b regularization paths for agePct12t29, pctWSocSec, pctUrban, agePct65up, and householdsize

8c)

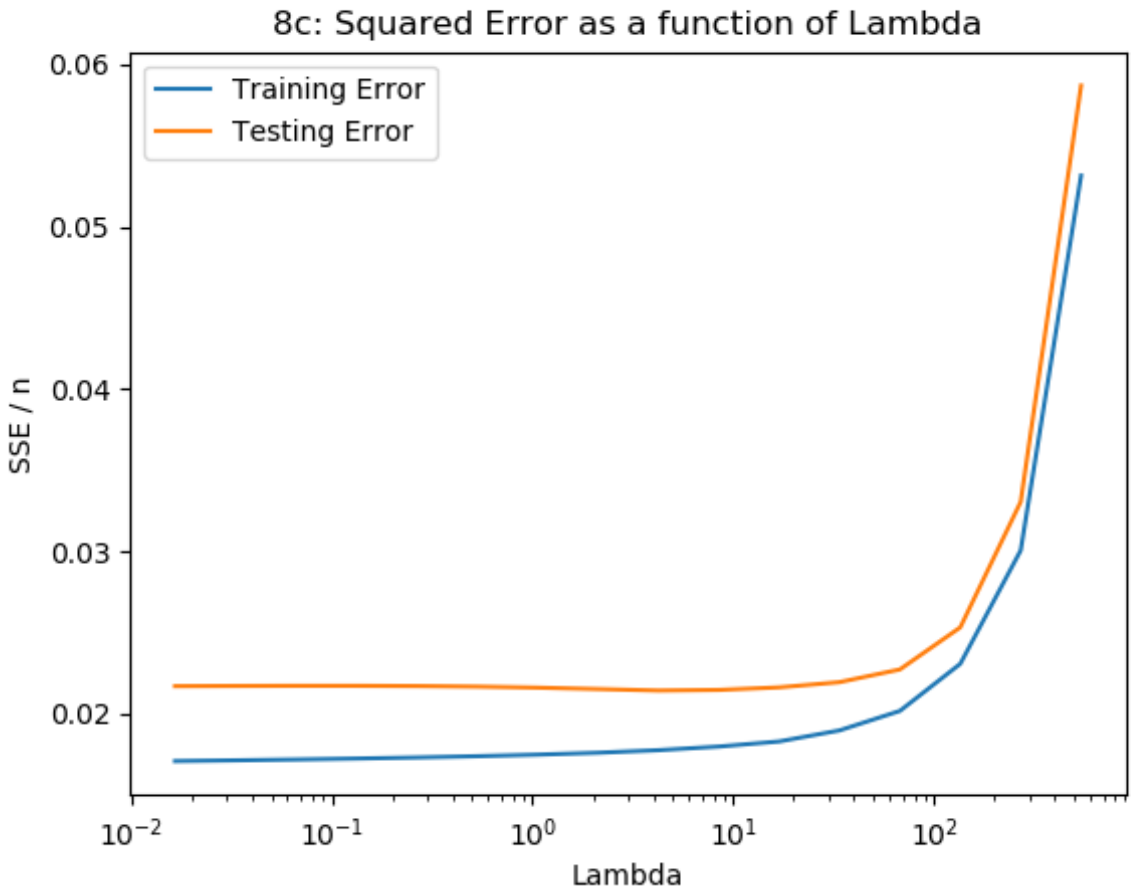

Figure 6: Problem 8c) Mean Squared Error (SSE / n) on the training and testing data sets for each value of λ of the L_1 regularization.

8d)

For this answer I used $\delta = 10^{-5}$ as my stopping condition.

The largest coefficient is around 0.068 associated to feature PctIlleg. This feature keeps track of the percentage of kids born to parents who never married. There is a positive correlation with this feature and the response variable (violent crime rate). This may imply having unmarried parents can foster an environment for the child that is more likely to lead to crime, but as we discuss in part e) below we cannot make any true conclusions about this: correlation does not imply causation.

The smallest coefficient is around -0.070 associated to the feature PctKids2Par. This feature keeps track of the percentage of kids in family housing with two parents. This may imply that having two parents can foster an environment where the child is less likely to lead to crime, but as we discuss in part e) below we cannot make any true conclusions about this: correlation does not imply causation.

Curiously, both these features are related to the child-parent relationship, suggesting this may be an important factor overall (or correlated with something important) in determining the amount of crime in the area.

8e)

Correlation does not imply causation. Regression says nothing about the causality of the variables. It only uses correlations in the data to construct meaningful predictions. It could be that the feature used in regression just has a high correlation with a feature outside the dataset which is actually meaningful in the predicting the response variable. For example, the large negative weight associated to the variable 'agePct65up' is probably not implying these 65+ year-olds are helping fighting crime. Rather the communities where 65+ year-olds are likely to live might have less crime because of some properties of those communities such as higher income, less young adults, etc. While this may only be a correlation, that does not mean our model is useless at making predictions about the crime in an area. Rather, it is only the interpretation of the coefficients that becomes ambiguous under this correlation.

3.1 Problem 8 Code

```
'''
lasso.py

implements the coordinate descent algorithm for LASSO regression.
LASSO promotes sparsity by regularizing the Least Squared Error
with an L1 norm.

X --> rows are data measurements, columns are specific features
y = vector of response variables

'''

import numpy as np
import matplotlib.pyplot as plt
import time


def generate_synthetic_data(n, d, k, variance, seed=123):
    '''
    Generates synthetic data following to procedure in the homework specification.
    n = number of data samples
    d = number of features per data sample
    k = number of relevant features (i.e. whose coefficient in w we expect to be non-negative)
    variance = variance for the zero-mean gaussian noise added on top of the true system
    seed = RNG seed for reproducibility

    '''
    if k > d:
        raise ValueError("The number of relevant features cannot be more than the number of
                                                    features")
    elif n < 1 or d < 1 or k < 0 or variance < 0:
        raise ValueError("Parameters n,d >= 1 and k,variance >= 0 must be true")

    w_true = np.zeros((d ,1))

    # j+1 is to account for zero-based indexing in python
    for j in range(k): w_true[j] = (j+1) / k

    np.random.seed(seed)
    X = np.random.normal(size = (n,d))
    errors = np.random.normal(scale = np.sqrt(variance), size=(n,))

    y = np.reshape(np.dot(w_true.T, X.T) + errors.T, (n,))
    return (X, y)



def min_null_lambda(X, y):
    '''
    Returns the smallest lambda value that generates a null solution
    (i.e. w is entirely zeros). Start with lambda at this value and
    decrease over time
    Assuming y is a column vector of responses and X is a matrix with each column a feature
    and each row a measurement (i.e. the data matrix)
    '''
    return 2*np.max(np.abs(np.dot(y.T - np.mean(y), X)))



def lasso_coordinate_descent(X, y, lam, w_init = None, delta = 10e-4):
    '''
    Runs the coordinate descent algorithm on the LASSO regression problem.

    X = data matrix with each measurement stored as a row (columns are features) n x features
    y = response variable : n x 1
    lam = lambda value used for regularization
    delta = stopping condition; if no element in w changes by more than delta in a single
                                                    iteration, stop.

    '''

    n = X.shape[0]
    d = X.shape[1]
    prev_w = np.ones((d,1))
```

```
    if w_init is None:
        w = np.zeros((d,))
    else:
        w = w_init

    c = np.zeros((d,))

    # a_k is just summing over all the rows of the squared entries. Thus, we can calculate it
    # quickly by squaring every entry in X (elementwise) then summing along the rows
    a = 2*np.sum(np.square(X), axis=0)

    # while we still have elements in w that changed by more than delta in last iteration
    while np.max(np.abs(w - prev_w)) > delta:

        prev_w = np.copy(w)

        wTXT = np.dot(w.T, X.T)

        b = 1/n * np.sum(y - wTXT)

        for k in range(d):

            # NOTE: Can use updated w_k values from SAME ROUND OF ITERATION piazza @199
            #       For this reason, we have to recalculate wTXT at every iteration
            c[k] = 2*np.dot(X[:, k], y - (b + np.dot(w.T, X.T) - w[k]*X[:, k]))

            if c[k] < -1*lam:
                w[k] = (c[k] + lam) / a[k]
            elif c[k] > lam:
                w[k] = (c[k] - lam) / a[k]
            else:
                w[k] = 0

    return (w, b)


# ================================================================================================
# Problem 8)


import pandas as pd
df_train = pd.read_table("data/crime-train.txt")
df_test = pd.read_table("data/crime-test.txt")

y_train = df_train["ViolentCrimesPerPop"].values
X_train = df_train.drop("ViolentCrimesPerPop", axis=1).values
y_test = df_test["ViolentCrimesPerPop"].values
X_test = df_test.drop("ViolentCrimesPerPop", axis=1).values


lam_max = min_null_lambda(X_train, y_train)
lam_ratio = 2 # factor to decrease lambda by after each iteration

current_lam = lam_max
lam_vals = []
delta = 10e-5 # threshold used to determine when to stop searching for optimal w
prev_w = None

# matrix with each column the weights vector generated by the corresponding lambda
W = None
B = []
while current_lam >= 0.01:

    lam_vals.append(current_lam)

    (w, b) = lasso_coordinate_descent(X_train, y_train, lam=current_lam, w_init=prev_w, delta=
                                                                    delta)

    if W is None:
        W = np.expand_dims(w, axis=1)
    else:
        W = np.concatenate((W, np.expand_dims(w, axis=1)), axis=1)

    B.append(b)

    prev_w = np.copy(w)
```

```
    current_lam /= lam_ratio

# Part a: Number non-zero entries vs lambda
plt.figure(3)
plt.semilogx(lam_vals, np.count_nonzero(W, axis=0), 'r-')
plt.xlabel('Lambda')
plt.ylabel('Nonzero Coefficients in w')
plt.title('8a: Nonzero coefficients vs Lambda')
#plt.show()

# Part b: Regularization Paths: agePct12t29, pctWSocSec, pctUrban, agePct65up, householdsize

# find where these columns reside; - 1 to account for the fact that the first column of
                                                df_train is our response
# variable y and thus has no associated weight
i1 = np.where(df_train.columns == "agePct12t29")[0] - 1
i2 = np.where(df_train.columns == "pctWSocSec")[0] - 1
i3 = np.where(df_train.columns == "pctUrban")[0] - 1
i4 = np.where(df_train.columns == "agePct65up")[0] - 1
i5 = np.where(df_train.columns == "householdsize")[0] - 1


k = len(lam_vals)

plt.figure(4)
plt.semilogx(lam_vals, np.reshape(W[i1, :], (k, )), \
             lam_vals, np.reshape(W[i2, :], (k,)), \
             lam_vals, np.reshape(W[i3, :], (k,)), \
             lam_vals, np.reshape(W[i4, :], (k,)), \
             lam_vals, np.reshape(W[i5, :], (k,)))

plt.xlabel('Lambda')
plt.ylabel('Coefficient Value')
plt.title('8b: Regularization Paths')
plt.legend(["agePct12t29", "pctWSocSec", "pctUrban", "agePct65up", "householdsize"])
#plt.show()

# Part c: Squared Error on training/test data

y_pred_train = np.dot(W.T, X_train.T) + np.expand_dims(B, axis=1)
SSE_train = 1/X_train.shape[0] * np.sum(np.square(y_pred_train - y_train), axis=1)
y_pred_test = np.dot(W.T, X_test.T) + np.expand_dims(B, axis=1)
SSE_test = 1/X_test.shape[0] * np.sum(np.square(y_pred_test - y_test), axis=1)

plt.figure(5)
plt.semilogx(lam_vals, SSE_train, lam_vals, SSE_test)
plt.legend(["Training Error", "Testing Error"])
plt.xlabel('Lambda')
plt.ylabel('SSE / n')
plt.title('8c: Squared Error as a function of Lambda')
plt.show()

# Part d:

(w30, _ ) = lasso_coordinate_descent(X_train, y_train, lam=30, delta=10e-5)

var_names = df_train.columns[1:] # skip the first varname corresponding to response variable
                                                ViolentCrimesPerPop
nonzero_coeffs = {w30[i]:var_names[i] for i in range(len(w30)) if not w30[i] == 0}
max_coeff = max(list(nonzero_coeffs.keys()))
min_coeff = min(list(nonzero_coeffs.keys()))
print(nonzero_coeffs)
print("feature with largest coefficient: ", nonzero_coeffs[max_coeff], " , value: ", max_coeff
                                                )
print("feature with smallest coefficient: ", nonzero_coeffs[min_coeff], " , value: ",
                                                min_coeff)
```

4 Programming: Binary Logistic Regression

4.1 Problem 9

9a)

Define:

$$\alpha = b + x_i^T w \rightarrow \frac{d\alpha}{dw_j} = x_{ij}$$

$$\beta = -y_i \alpha \rightarrow \frac{d\beta}{d\alpha} = -y_i$$

$$\kappa = 1 + e^{\beta} \rightarrow \frac{d\kappa}{d\beta} = e^{\beta}$$

$$\Omega = log(\kappa) = log(1 + exp(-y_i(b + x_i^T w)))) \rightarrow \frac{d\Omega}{d\kappa} = \frac{1}{\kappa}$$

Using these definitions, we can calculate $\nabla_w J(w, b)$ by considering taking the derivative with respect to w_i, applying chain rule, and just repeating this for every entry in w.

$$\frac{dJ(w,b)}{dw_j} = \frac{1}{n}\sum_{i=1}^{n} \frac{d\Omega}{d\kappa}\frac{d\kappa}{d\beta}\frac{d\alpha}{dw_j} + 2\lambda w_j$$

$$= \frac{1}{n}\sum_{i=1}^{n} \frac{1}{1 + exp(-y_i(b + x_i^T w))} e^{-y_i(b+x_i^T w)}(-y_i)(x_{ij}) + 2\lambda w_j$$

Note that if $\mu_i(w, b) = \frac{1}{1+exp(-y_i(b+x_i^T w))}$ then we can see:

$$exp(-y_i(b + x_i^T w)) = \frac{1 - \mu_i(w, b)}{\mu_i(w, b)}$$

Thus we can simplify the above expression to:

$$= \frac{1}{n}\sum_{i=1}^{n} \mu_i(w, b)\frac{1 - \mu_i(w, b)}{\mu_i(w, b)}(-y_i x_{ij}) + 2\lambda w_j$$

$$\frac{dJ(w,b)}{dw_j} = \frac{1}{n}\sum_{i=1}^{n} -y_i x_{ij}(\mu_i(w, b) - 1) + 2\lambda w_j$$

Combining each of these derivatives into vector form this yields:

$$\boxed{\nabla_w J(w, b) = \frac{1}{n}\sum_{i=1}^{n} (-y_i x_i(1 - \mu_i(w, b))) + 2\lambda w}$$

When we are calculating $\nabla_b J(w, b)$ only a few things change. Now $\frac{d\alpha}{db} = 1$ and the term $\lambda||w||_2^2$ goes to zero as it does not include b. This yields:

$$\frac{dJ(w,b)}{db} = \frac{1}{n}\sum_{i=1}^{n} \frac{d\Omega}{d\kappa}\frac{d\kappa}{d\beta}\frac{d\alpha}{db}$$

$$= \frac{1}{n}\sum_{i=1}^{n} \frac{1}{1 + exp(-y_i(b + x_i^T w))} e^{-y_i(b+x_i^T w)}(-y_i)$$

$$= \frac{1}{n}\sum_{i=1}^{n} \mu_i(w, b)\frac{1 - \mu_i(w, b)}{\mu_i(w, b)}(-y_i)$$

$$\boxed{\nabla_b J(w, f) = \frac{1}{n}\sum_{i=1}^{n} -y_i(1 - \mu_i(w, b))}$$

9bi)

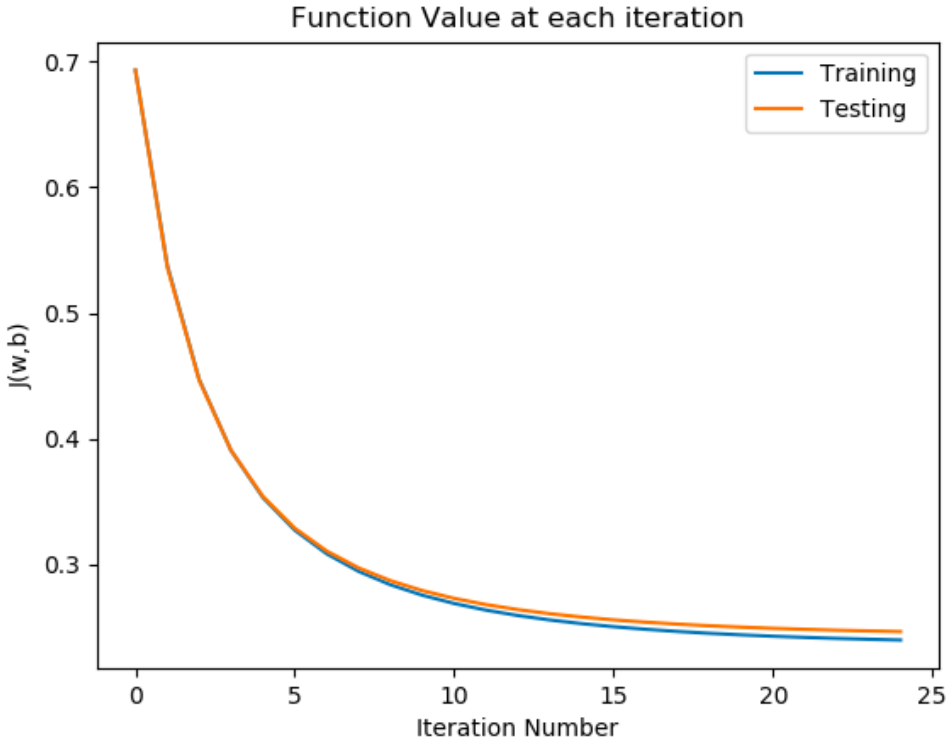

Figure 7: Problem 9bi) Gradient Descent on Binary Logistic Regression

9b ii)

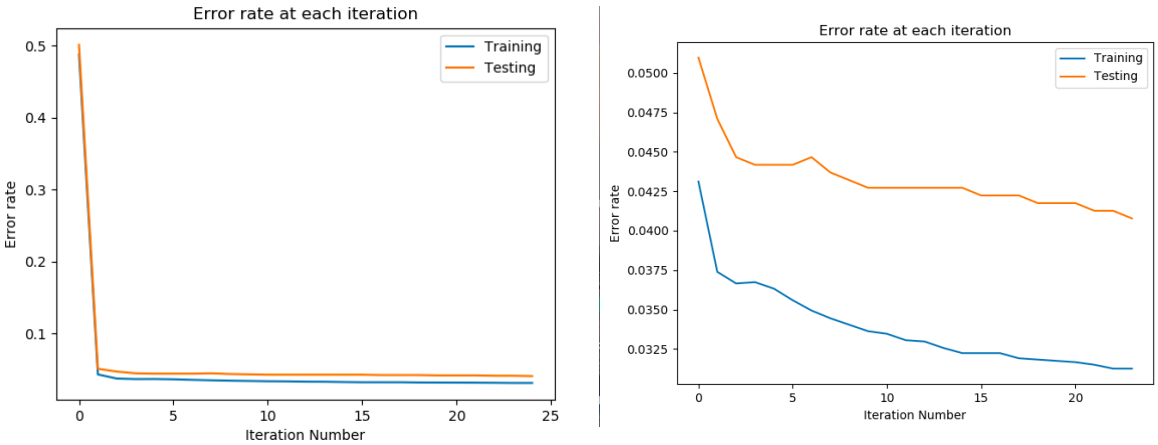

Figure 8: Problem 9b ii). In the left, we can see that the error rate starts at around 50% which is exactly what we would expect. With no training, we are simply guessing at the true value, giving us a 50-50 chance of getting it correct in this binary system (assuming 2's and 7's were evenly represented). The right image is a zoomed in version of the left. We can still see the error is decreasing over each iteration but seems to be converging.

In the below figures for Stochastic Gradient descent, problems 9c/d, we observe noisy convergence. Here, the iteration number counts the number of updates performed on w rather than the number of cycles through the dataset.

9ci)

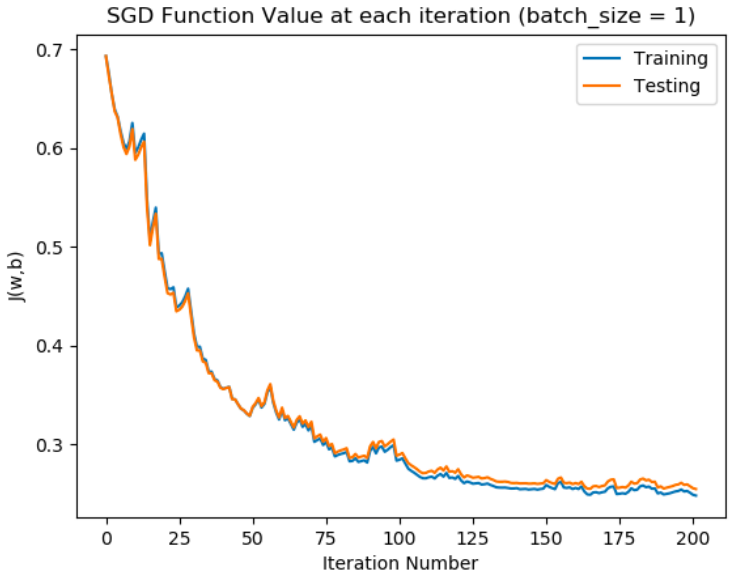

Figure 9: Problem 9ci) Stochastic Gradient Descent with a batch size of one. Due to the small batch size, we can see fairly noisy convergence.

9cii)

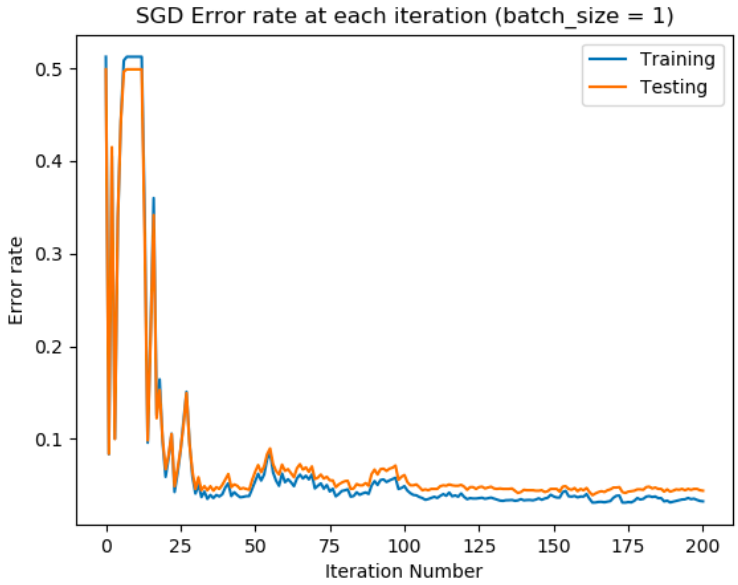

Figure 10: Problem 9c ii) In the left, we can see that the error rate starts at around 50% which is exactly what we would expect. With no training, we are simply guessing at the true value, giving us a 50-50 chance of getting it correct in this binary system (assuming 2's and 7's were evenly represented). The right image is a zoomed in version of the left. We can still see the error is decreasing over each iteration. As in above, we observe fairly noisy convergence due to the small batch size.

9di)

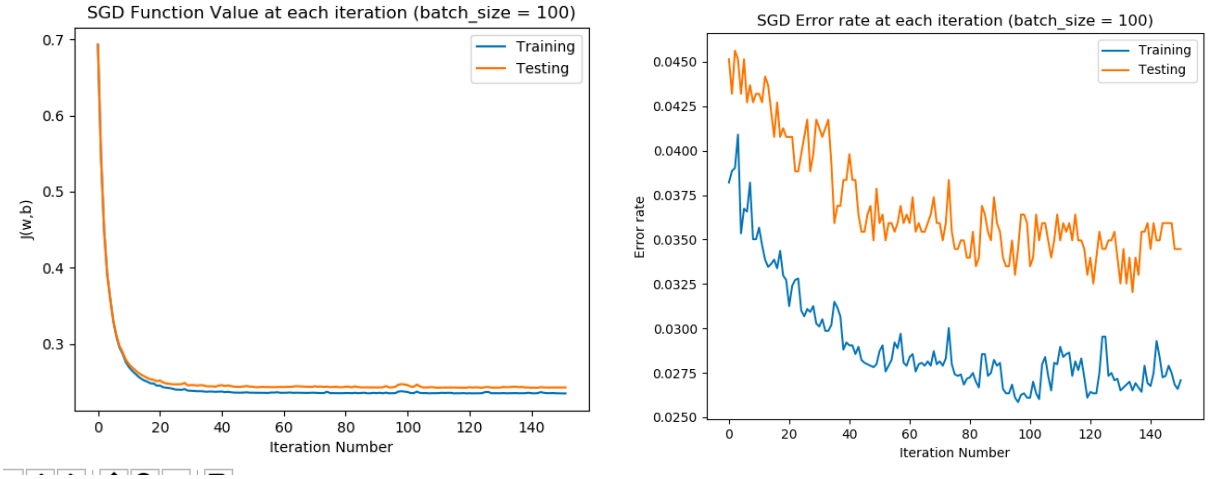

Figure 11: Problem 9d i). Stochastic Gradient Descent with a batch size of 100. We can see noisy convergence, but less so than before due to the larger batch size. The right is a zoomed in version of the left to make the noisy convergence more obvious (the $x - axis$ on the right begins at iteration 20 on the left graph)

9dii)

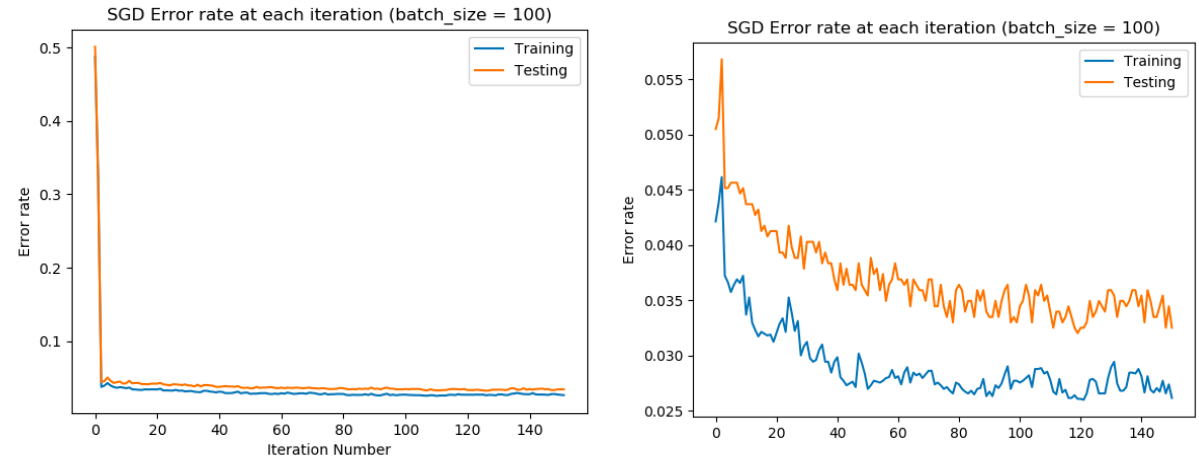

Figure 12: Problem 9d ii). In the left, we can see that the error rate starts at around 50% which is exactly what we would expect. With no training, we are simply guessing at the true value, giving us a 50-50 chance of getting it correct in this binary system (assuming 2's and 7's were evenly represented). The right image is a zoomed in version of the left. We can still see the error is decreasing over each iteration. As in above, we observe noisy convergence, but less noise than before due to the larger batch size.

4.2 Problem 9 Code

```
'''
logistic.py

'''

import numpy as np
import random
import matplotlib.pyplot as plt
from mnist.loader import MNIST

def grad_J(X, y, w, b, lam):
    '''
    returns a vector that represents the gradient of J at the given points w,b

    X = data matrix with rows as measurements and columns are features (n x d)
    y = response variable (n x 1)
    w = weights vector (d x 1)
    b = bias term (1 x 1)
    lam = lambda value used for regularization
    '''

    n = y.size

    # y is a (n, ) dim vector instead of a (n, 1) vector. We adjust the dims so the
                                                        elementwise product
    # performed below behaves as expected
    y = np.expand_dims(y, axis=1)

    temp = b + np.dot(w.T, X.T)

    # temp.T as current temp is a row vector
    mu = 1 / (1 + np.exp(np.multiply(-1*y, temp.T)))

    nabla_w = 1/n * np.dot(np.multiply(-y, X).T, 1-mu) + 2 * lam * w
    nabla_b = 1/n * np.dot(-y.T, 1-mu)

    return np.vstack((nabla_w, nabla_b))

def J_function(X, y, w,b, lam):
    '''
    Calculates the value of the J(w,b) the (normalized) logistic error function

    X = data matrix with rows as measurements and columns are features (n x d)
    y = response variable (n x 1)
    w = weights vector (d x 1)
    b = bias term (1 x 1)
    lam = lambda value used for regularization
    '''

    inside = np.multiply(-y, b + np.dot(w.T, X.T))
    n = y.size

    # squeeze removes the unnecessary dimensions: i.e. (25,1,1) --> (25,)
    return np.squeeze(1/n * np.sum(np.log(1 + np.exp(inside))) + lam * np.dot(w.T, w))

def error_rate(X, y, w, b):
    '''
    Uses the given weights w and bias b to make a classification sign(wTx + b) for each data
    measurement x in X (these measurements are stored as rows). Compares these against the
                                                        true
    label stored in y to calculate an error rate.

    X = data matrix with rows as measurements and columns are features (n x d)
    y = response variable (n x 1)
    w = weights vector (d x 1)
    b = bias term (1 x 1)
    lam = lambda value used for regularization
    '''

    n = y.size
    predictions = np.sign(b + np.dot(w.T, X.T))
```

```
    # since np.sign(0) returns zero, we have to choose to set these to 1 or -1. Arbitrarily,
                                                    we choose 1
    # Here if the element is zero, we replace it with 1, else we take whatever was in
                                                    predictions.
    predictions = np.where(predictions == 0, 1, predictions)

    # NOTE: if we add the predictions and the labels, each correct prediction will be either -
                                                    2 or 2 and each
    # incorrect prediction will be zero as all labels/guesses are -1 or 1. Thus the error rate
                                                     will be:
    #  1 - sum of abs((predictions + labels)) / (2*total)
    error = 1 - np.sum( np.abs(predictions + y) ) / (2*n)
    return error


def gradient_descent(x_init, gradient_function, eta=0.1, delta=10e-4):
    '''
    Runs gradient descent to calculate minimizer x of the function whose gradient
    is defined by the given gradient_function.

    x_init = [w,b] is the initial values to set to the vector being descended on (in this
                                                    problem w and b)
    gradient_function = a function that takes in a vector x and outputs gradient evaluated at
                                                    that point
    eta  = the learning rate for gradient descent
    delta = stopping condition; stop if all entries in gradient change by less than delta in
                                                    an iteration.

    '''
    x = x_init
    all_xs = [x]
    grad = gradient_function(x)
    while np.max(np.abs(grad)) > delta:
        # perform a step in gradient descent

        x = x - eta * grad
        grad = gradient_function(x)
        all_xs.append(x)

    # x is the best variable values; all_xs shows x value at each iteration
    return (x, all_xs)


def SGD(X, y, x_init, gradient_function, batch_size, eta=0.1, iteration_num=100):
    '''
    Runs STOCHASTIC gradient descent to calculate minimizer x of the given function whose
    gradient is defined by gradient_function.

    x_init = [w,b] is the initial values to set to the vector being descended on (in this
                                                    problem w and b)
    gradient_function = a function that takes in a vector x and outputs gradient evaluated at
                                                    that point
    batch_size = size of mini-batches used for SGD to approximate gradient across whole
                                                    dataset
    eta  = the learning rate for gradient descent
    iteration_num = number of iterations (updates of x) to perform before returning the
                                                    current x
    '''

    x = x_init
    n = y.size
    all_xs = [x]

    itr = 0
    while itr < iteration_num:
        # Loop through full dataset once, performing a round of updates

        train_order = list(range(n))
        random.shuffle(train_order)

        for batch_num in range(n // batch_size):

            itr += 1
            sample = train_order[batch_num * batch_size : (batch_num + 1) * batch_size]
            X_batch = X[sample, :]
            y_batch = y[sample]
```

```
            grad = gradient_function(X_batch=X_batch, y_batch=y_batch, x=x)
            x = x - eta * grad

            all_xs.append(x)
            if itr > iteration_num: break


        # handle case where there is one uneven batch left
        else:
            if n % batch_size != 0:
                itr += 1
                sample = train_order[(n // batch_size) * batch_size : ]
                X_batch = X[sample ,:]
                y_batch = y[sample]

                grad = gradient_function(X_batch, y_batch, x)
                x = x - eta * grad

                all_xs.append(x)

    return (x, all_xs)



# ======================================================================

# Load MNIST Data and filter for 2's and 7's
mndata = MNIST('./data')
X_train, labels_train = map(np.array, mndata.load_training())
X_test, labels_test = map(np.array, mndata.load_testing())
X_train = X_train / 255.0
X_test = X_test / 255.0

# Filter out all digits besides two or seven
X_train = X_train[(labels_train == 2) + (labels_train == 7), :]
X_test = X_test[(labels_test == 2) + (labels_test == 7), :]

# filter out all digits besides two and seven. Convert label
# for a 7 to 1 and a 2 to -1
Y_train = labels_train[(labels_train == 2) + (labels_train == 7)]
Y_train = np.where(Y_train == 7, 1, -1)
Y_test = labels_test[(labels_test == 2) + (labels_test == 7)]
Y_test = np.where(Y_test == 7, 1, -1)

# ======================================================================

n = X_train.shape[0]
d = X_train.shape[1] # 28 x 28 = 784
lam = 0.1

# define initial vector and the gradient function
x_init = np.zeros((d+1, 1)) # = [w, b]
gradient_function_train = lambda x: grad_J(X_train, Y_train, x[:-1], x[-1], lam)
delta = 0.01
eta = 0.1 # learning rate

(x_best_train, all_xs_train) = gradient_descent(x_init, gradient_function_train, eta, delta)


# Problem 9b part i)

plt.figure(1)
plt.plot([J_function(X=X_train, y=Y_train, w=x[:-1], b=x[-1], lam=lam) for x in all_xs_train])
plt.plot([J_function(X=X_test, y=Y_test, w=x[:-1], b=x[-1], lam=lam) for x in all_xs_train]) #
                                                    use parameters learned from training dataset
plt.title('Function Value at each iteration')
plt.ylabel('J(w,b)')
plt.xlabel('Iteration Number')
plt.legend(['Training', 'Testing'])
#plt.show()

# Problem 9b part ii)

plt.figure(2)
# skips plotting the error rate at iteration zero with w = 0
```

```
plt.plot([error_rate(X=X_train, y=Y_train, w=x[:-1], b=x[-1]) for x in all_xs_train[1:]])
plt.plot([error_rate(X=X_test, y=Y_test, w=x[:-1], b=x[-1]) for x in all_xs_train[1:]]) # use
                                                    parameters learned from training dataset
plt.title('Error rate at each iteration')
plt.ylabel('Error rate')
plt.xlabel('Iteration Number')
plt.legend(['Training', 'Testing'])
plt.show()



# ================================================================================

# Problem 9c)

n = X_train.shape[0]
d = X_train.shape[1] # 28 x 28 = 784
x_init = np.zeros((d+1, 1)) # = [w, b]
batch_size = 1
eta = 0.01 # learning rate 0.00005, delta = 0.01
lam = 0.1

num_iterations = 200

# This time, the gradient function accepts some data as an input (i.e. the batch)
sgd_grad_function = lambda X_batch, y_batch, x: grad_J(X_batch, y_batch, x[:-1], x[-1], lam)
(x_best_sgd, all_xs_sgd) = SGD(X=X_train, y=Y_train, x_init=x_init, gradient_function=
                                                    sgd_grad_function, \
                                batch_size=batch_size, eta=eta, iteration_num=num_iterations)

# 9ci)

plt.figure(3)
plt.plot([J_function(X=X_train, y=Y_train, w=x[:-1], b=x[-1], lam=lam) for x in all_xs_sgd])
plt.plot([J_function(X=X_test, y=Y_test, w=x[:-1], b=x[-1], lam=lam) for x in all_xs_sgd]) #
                                                    use parameters learned from training dataset
plt.title('SGD Function Value at each iteration (batch_size = 1)')
plt.ylabel('J(w,b)')
plt.xlabel('Iteration Number')
plt.legend(['Training', 'Testing'])
#plt.show()


# 9cii)

plt.figure(4)
# skips plotting the error rate at iteration zero with w = 0
plt.plot([error_rate(X=X_train, y=Y_train, w=x[:-1], b=x[-1]) for x in all_xs_sgd[1:]])
plt.plot([error_rate(X=X_test, y=Y_test, w=x[:-1], b=x[-1]) for x in all_xs_sgd[1:]]) # use
                                                    parameters learned from training dataset
plt.title('SGD Error rate at each iteration (batch_size = 1)')
plt.ylabel('Error rate')
plt.xlabel('Iteration Number')
plt.legend(['Training', 'Testing'])
plt.show()


# ================================================================================

# Problem 9d)
n = X_train.shape[0]
d = X_train.shape[1] # 28 x 28 = 784
x_init = np.zeros((d+1, 1)) # = [w, b]
batch_size = 100
eta = 0.1 # learning rate 0.00005, delta = 0.01
lam = 0.1

num_iterations = 150


# This time, the gradient function accepts some data as an input (i.e. the batch)
sgd_grad_function = lambda X_batch, y_batch, x: grad_J(X_batch, y_batch, x[:-1], x[-1], lam)
(x_best_sgd, all_xs_sgd100) = SGD(X=X_train, y=Y_train, x_init=x_init, gradient_function=
                                                    sgd_grad_function, \
                                  batch_size=batch_size, eta=eta, iteration_num=num_iterations
                                                                                        )
```

```
# 9di)

plt.figure(5)
plt.plot([J_function(X=X_train, y=Y_train, w=x[:-1], b=x[-1], lam=lam) for x in all_xs_sgd100[
                                                    20:]])
plt.plot([J_function(X=X_test, y=Y_test, w=x[:-1], b=x[-1], lam=lam) for x in all_xs_sgd100[20
                                                    :]]) # use parameters learned from training
                                                    dataset
plt.title('SGD Function Value at each iteration (batch_size = 100)')
plt.ylabel('J(w,b)')
plt.xlabel('Iteration Number')
plt.legend(['Training', 'Testing'])
#plt.show()


# 9dii)

plt.figure(6)
# skips plotting the error rate at iteration zero with w = 0
plt.plot([error_rate(X=X_train, y=Y_train, w=x[:-1], b=x[-1]) for x in all_xs_sgd100[1:]])
plt.plot([error_rate(X=X_test, y=Y_test, w=x[:-1], b=x[-1]) for x in all_xs_sgd100[1:]]) # use
                                                     parameters learned from training dataset
plt.title('SGD Error rate at each iteration (batch_size = 100)')
plt.ylabel('Error rate')
plt.xlabel('Iteration Number')
plt.legend(['Training', 'Testing'])
plt.show()
```