Homework #2

Spring 2020, CSE 546: Machine Learning

Roman Levin
1721898

Collaborators: compared answers with Tyler Chen, Diya Sashidhar, Katherine Owens

Conceptual Questions

A.0 **Solution:**

a. **No,** because there could be several perfectly correlated features, each with large positive weight (e.g. 'number of bathrooms' and 'number of showers'). If they are perfectly correlated, then either one could be removed without compromising the quality of the predictions.

b. Because of **the shape of L1 and L2 norm balls (level sets)**. The L1 ball is more "spiky" (with sparse vertices) and therefore level sets of the original objective are more likely to intersect it at its sparse vertices (unlike in the case of L2) resulting in a sparse solution to the regularized problem. Also, L0 penalty explicitly penalizes the number of non-zero elements and, between $p = 2$ and $p = 1$, the latter is closer to $p = 0$, and thus L1 norm is a better proxy for L0 than L2 is.

c. Upside: promotes **sparsity** even better than L1. Downside: **non-convex**.

d. **True**

e. Because **in expectation it goes in the right direction.**

f. Advantage: one iteration of SGD **is much less expensive computationally** compared to GD. Disadvantage: SGD requires **more iterations** to reach the same error.

Convexity and Norms

A.1 **Solution:**

a. First, let's prove the following (obvious) fact: $\forall a, b \in \mathbb{R} : |a + b| \leq |a| + |b|$:

$|a+b|^2 = a^2 + b^2 + 2ab \leq a^2 + b^2 + 2|a||b| = (|a| + |b|)^2 \Rightarrow |a + b| \leq |a| + |b|$ since $|a+b| \geq 0, |a| + |b| \geq 0.\square$

Now, let's check the definition of the norm for L1 norm $f(x) = \sum_{i=1}^n |x_i|$:

- (Non-negativity): $f(x) \geq 0$ since $\forall i : |x_i| \geq 0$. Now, $|x_i| = 0$ iff $x_i = 0$, so $f(x) = 0$ iff $x = 0$. $\square$
- (Absolute scalability): $\forall a \in \mathbb{R}, x \in \mathbb{R}^n : f(ax) = \sum_{i=1}^n |ax_i| = \sum_{i=1}^n |a||x_i| = |a|f(x).\square$
- (Triangle inequality): $\forall x, y \in \mathbb{R}^n : f(x)+f(y) = \sum_{i=1}^n |x_i|+\sum_{i=1}^n |y_i| = \sum_{i=1}^n |x_i|+|y_i| \geq \sum_{i=1}^n |x_i + y_i| = f(x + y)$, where the second to last inequality follows from the triangle inequality for absolute values on $\mathbb{R}$ above. $\square$

b. Consider $x = [0,1]^T, y = [1,0]^T$. Then $g(x)+g(y) = 1^2+1^2 = 2, g(x+y) = (1+1)^2 = 4$, so for these x, y: $g(x+y) > g(x)+g(y)$ and the triangle inequality does not hold which means g is not a norm. $\square$

B.1 **Solution:**

- Note that $\max_i(|x_i|^2) = (\max_i |x_i|)^2$ since $|x_i| \geq 0$. Now

$$\|x\|_2^2 = \sum_i |x_i|^2 \geq \max_i(|x_i|^2) = (\max_i |x_i|)^2 = \|x\|_\infty^2 \Leftrightarrow \|x\|_2 \geq \|x\|_\infty \text{ since } \|x\|_2 \geq 0, \|x\|_\infty \geq 0.$$

- $\|x\|_1^2 = (\sum_i |x_i|)^2 = \sum_i |x_i|^2 + \underbrace{\sum_{i\neq j} |x_i||x_j|}_{\geq 0} \geq \sum_i |x_i|^2 = \|x\|_2^2 \Leftrightarrow \|x\|_1 \geq \|x\|_2$ since $\|x\|_2 \geq 0, \|x\|_1 \geq 0$

- That is, $\boxed{\|x\|_1 \geq \|x\|_2 \geq \|x\|_\infty . \square}$

A.2 **Solution:**

- **I is not convex**: line segment between points b and c is not in the set, while the points are.
- **II is convex.**
- **III is not convex**: line segment between points d and a is not in the set, while the points are.

A.3 **Solution:**

a. **I is convex** on $[a, c[$.

b. **II is not convex** on $[a, c]$: for $\lambda = 1/2$: $f(\lambda b + (1-\lambda)c) > \lambda f(b) + (1-\lambda) f(c)$.

c. **III is convex** on $[a, d]$: for $\lambda = 1/2$: $f(\lambda a + (1-\lambda)c) > \lambda f(a) + (1-\lambda) f(c)$

d. **III is convex** on $[c, d]$.

B.2 **Solution:**

a. We will show convexity by definition. Take any $\lambda \in [0,1]$. Note that $(1-\lambda) \in [0,1]$. Then, taking any $x, y \in \mathbb{R}^n$, by triangle inequality and absolute scalability:

$$\|\lambda x + (1-\lambda)y\| \leq \|\lambda x\| + \|(1-\lambda)y\| \leq \lambda\|x\| + (1-\lambda)\|y\| \square.$$

b. $B := \{x \in \mathbb{R}^n : \|x\| \leq 1\}$. Take any $\lambda \in [0,1]$ and any $x, y \in B$. Then (using derivations in a.):

$$\|\lambda x + (1-\lambda)y\| \leq \lambda \underbrace{\|x\|}_{\leq 1 \text{ for } x \in B} + (1-\lambda) \underbrace{\|y\|}_{\leq 1 \text{ for } y \in B} \leq \lambda + 1 - \lambda = 1 \Rightarrow \lambda x + (1-\lambda)y \in B$$

So B is indeed a convex set. $\square$

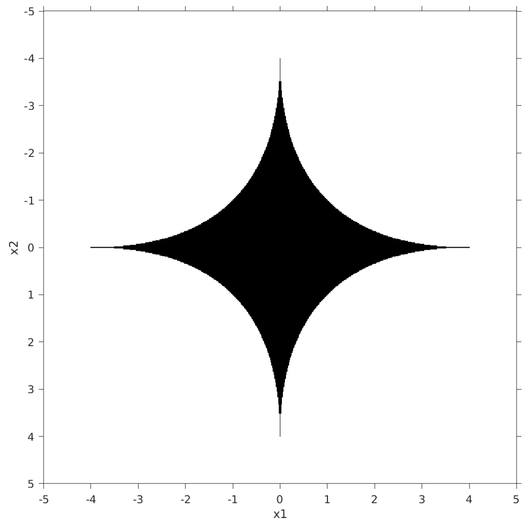

Figure 1: Plot of $\{(x,y) : g(x,y) \leq 4\}$, where $g(x,y) = (|x|^{1/2} + |y|^{1/2})^2$

c. The plot of $L := \{(x,y) : g(x,y) \leq 4\}$, where $g(x,y) = (|x|^{1/2} + |y|^{1/2})^2$ is on Figure 1. This set is not convex because for $\lambda = 0.5$ and points $x = [0,-4]^T, y = [4,0]^T, x, y \in L$:

$$\lambda x + (1-\lambda)y \notin L\square$$

B.3 **Solution:**

a.
- Let's first show that the sum of convex functions is convex. Let f, g be convex, consider $f(x)+g(x)$. Take any $\lambda \in [0,1]$, take any x, y. Then, by convexity of f and g:

  $$f(\lambda x+(1-\lambda)y)+g(\lambda x+(1-\lambda)y) \leq \lambda f(x)+(1-\lambda)f(y)+\lambda g(x)+(1-\lambda)g(y) = \lambda(f(x)+g(x))+(1-\lambda)(f(y)+g(y)).\square$$

  That is, $f(x)+g(x)$ is convex if $f(x)$ and $g(x)$ are convex. Now, a sum $\sum_{i=1}^n f_i(x)$ of n convex functions $f_i(x)$ is also convex because by the above we can first show that $f_1(x)+f_2(x)$ is convex, that implies the convexity of $f_1(x)+f_2(x)+f_3(x)$, etc. At every step j, we know that $\sum_{i=1}^j f_i(x)$ is convex and $f_{j+1}(x)$ is also convex, so $\sum_{i=1}^{j+1} f_i(x)$ is convex. That is, by induction, it follows that $\sum_{i=1}^n f_i(x)$ is convex. $\square$
- Now, multiplication by $\lambda > 0$ obviously preserves convexity and thus $\lambda\|w\|$ is convex because $\|w\|$ is convex. To see that, take any $\alpha \in [0,1]$ and any x, y:

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

- Finally, since $l_i(w)$ are convex $\Rightarrow \sum_{i=1}^n l_i(w)$ is convex. As shown above, $\lambda\|w\|$ is convex too. So $\sum_{i=1}^n l_i(w) + \lambda\|w\|$ is convex as a sum of two convex functions. $\square$

b. Because convexity implies that every local minimum is a global minimum. $\square$

Lasso

A.4 **Solution:**

a. See Plot 1 in Figure 2 (left). Note that for this problem I treated numbers less than 1e-14 in absolute value as zeros.

b. See Plot 2 in Figure 2 (right).

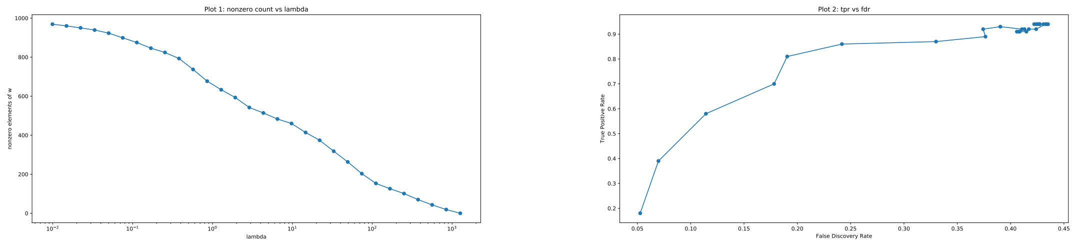

Figure 2: Problem A4. Left: A4.a, Right: A4.b

c. From the plots we see, as expected, that greater λ forces more weights to zero. That is, increasing λ results in a more sparse solution. However, in Plot 2, we see that too large λ results in zero solution which gives poor performance. If we use λ which is too small, we end up in another extreme, where we have a high false discovery rate. One should choose λ which is close to the upper left corner on Plot 2. (Probably, making this plot on a validation set would be a good idea too for choosing λ).

```
######################
#Problem A4, HW2
######################
import numpy as np
import scipy
import matplotlib.pyplot as plt

class Lasso:

        def __init__(self, reg_lambda=1E-8):
                """
                Constructor
                """
                self.reg_lambda = reg_lambda
                self.w = None
                self.b = None
                self.conv_history = None

        def objective(self, X, y):
                """
                Returns Lasso objective value

                Parameters
                ----------
                X : np.array of shape (n,d)
                        Features
                y : np.array of shape (n,)
                        Labels
                Returns
                -------
                float
                        Objective value for current w, b and given reg_lambda
                """
```

```
        return (np.linalg.norm(X.dot(self.w) + self.b -
                y))**2 + self.reg_lambda*np.linalg.norm(self.w, ord = 1)

def fit(self, X, y, w_init = None, delta = 1e-4):
        """
                Trains the Lasso model
        Parameters
        ----------
        X : np.array of shape (n,d)
                Features
        y : np.array of shape (n,)
                Labels
        w_init : np.array of shape (d,)
                Initial guess for w
        delta : float
                Stopping criterion

        Returns
        -------
        convergence_history : array
                convergence history
        """
        iter_count = 0
        n, d = X.shape
        if w_init is None:
                w_init = np.zeros(d)
        w_prev = w_init + np.inf #just to enter the loop
        self.w = w_init
        convergence_history = []
        a = 2*np.sum(X**2, axis = 0) #precompute a
        print('shape a:', a.shape, 'should be ', d)
        while np.linalg.norm(self.w-w_prev, ord = np.inf) >= delta:
                iter_count += 1
                w_prev = np.copy(self.w)
                self.b = np.mean(y - X.dot(self.w))
                for k in range(d):
                        not_k_cols = np.arange(d) != k
                        a_k = a[k]
                        c_k = 2*np.sum(X[:,k]*(y - (self.b
                                + X[:, not_k_cols].dot(self.w[not_k_cols]))), axis = 0)
                        self.w[k] = np.float(np.piecewise(c_k,
                                [c_k < -self.reg_lambda, c_k > self.reg_lambda, ],
                                [(c_k+self.reg_lambda)/a_k, (c_k-self.reg_lambda)/a_k, 0]))
                if iter_count % 1 == 0:
                        print('Iter ', iter_count, ' Loss:', self.objective(X,y))
                convergence_history.append(self.objective(X,y))
        self.conv_history = convergence_history
        print('converged in: ', len(convergence_history))
        return convergence_history


def predict(self, X):
        """
        Use the trained model to predict values for each instance in X
        Arguments:
```

```python
                X is a n-by-d numpy array
        Returns:
                an n-by-1 numpy array of the predictions
        """
        return X.dot(self.w) + self.b



def generate_synthetic_data(n, d, k, sigma):
    """
            Generates the synthetic dataset
    Parameters
    ----------
    n : float
    d : float
    k : float
    sigma : float

    Returns
    -------
    X : np.array of shape (n,d)
    y : np.array of shape (n,)
    w : np.array of shape (d,)
    """
    #Create true w:
    w = np.arange(d)/k
    w[k+1:] = 0

    #Draw X at random:
    X = np.random.normal(loc=0.0, scale=1.0, size=(n,d))
    #Generate y:
    eps = np.random.normal(loc=0.0, scale=sigma, size=(n,))
    y = X.dot(w) + eps
    return X, y, w


if __name__ == "__main__":
    #Generate synthetic data:
    n = 500
    d = 1000
    k = 100
    sigma = 1
    X, y, w_true = generate_synthetic_data(n, d, k, sigma)

    nonzeros = []
    tpr = []
    fdr = []
    lambda_max = np.max(np.sum(2*X*(y - np.mean(y))[:, None], axis = 0))
    print(lambda_max)
    lambdas = [lambda_max/(1.5**i) for i in range(30)]
    w_init = None
    for reg_lambda in lambdas:
        model = Lasso(reg_lambda = reg_lambda)
        model.fit(X,y, w_init, delta = 1e-3)
        w_init = np.copy(model.w)
```

```
            total_num_of_nonzeros = np.sum(abs(model.w) > 1e-14)
            number_of_incorrect_nonzeros = np.sum(model.w[abs(w_true) <= 1e-14] > 1e-14)
            number_of_correct_nonzeros = np.sum(model.w[abs(w_true) > 1e-14] > 1e-14)
            nonzeros.append(total_num_of_nonzeros)
            fdr.append(number_of_incorrect_nonzeros/total_num_of_nonzeros)
            tpr.append(number_of_correct_nonzeros/k)
            print('Current nonzero number:', np.sum(abs(model.w) > 1e-14))

    #Part a
    plt.figure(figsize = (15,7))
    plt.plot(lambdas, nonzeros, '-o')
    plt.xscale('log')
    plt.title('Plot 1: nonzero count vs lambda')
    plt.xlabel('lambda')
    plt.ylabel('nonzero elements of w')
    plt.savefig('figures/A4a.pdf')
    plt.show()

    #Part b
    plt.figure(figsize = (15,7))
    plt.plot(fdr, tpr, '-o')
    plt.title('Plot 2: tpr vs fdr')
    plt.xlabel('False Discovery Rate')
    plt.ylabel('True Positive Rate')
    plt.savefig('figures/A4b.pdf')
    plt.show()
```

Code for A4

A.5 **Solution:**

a. See Figure 3 (top left).

b. See Figure 3 (top right).

c. See Figure 3 (bottom).

d. The most positive weight: **PctIlleg**
The most negative weight: **PctKids2Par**
That is, the crime rate has the most positive correlation with PctIlleg (percentage of kids born to never married). On the other hand side, the crime rate has the most negative correlation with PctKids2Par (percentage of kids in family housing with two parents).

e. Correlation is not the same with causality. Just because there are fewer people over 65 in the high crime areas, does not mean that the number of people over 65 decreases the crime rate. It could also mean that people over 65 tend to move out from high crime areas. Just like firetrucks don't cause fires. Seeing a fire truck is only correlated with seeing a burning building and it is the fire which causes the presence of a firetruck.

```
######################
#Problem A5, HW2
######################

import numpy as np
import scipy
import matplotlib.pyplot as plt
import pandas as pd
```

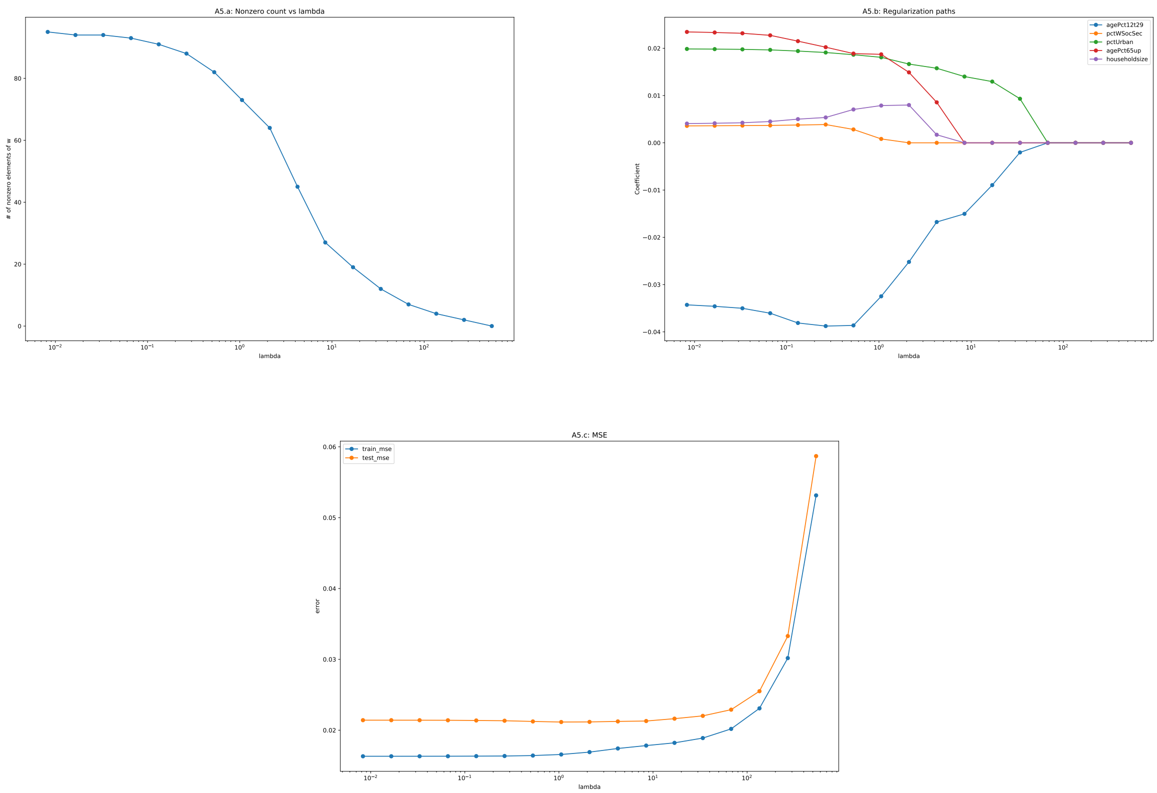

Figure 3: Problem A5. Top Left: A5.a, Top Right: A5.b, Bottom: A5.c

```
class Lasso:

    def __init__(self, reg_lambda=1E-8):
        """
        Constructor
        """
        self.reg_lambda = reg_lambda
        self.w = None
        self.b = None
        self.conv_history = None

    def objective(self, X, y):
        """
        Returns Lasso objective value

        Parameters
        ----------
        X : np.array of shape (n,d)
                Features
        y : np.array of shape (n,)
                Labels
        Returns
        -------
        float
```

```
                Objective value for current w, b and given reg_lambda
        """
        return (np.linalg.norm(X.dot(self.w) + self.b
                - y))**2 + self.reg_lambda*np.linalg.norm(self.w, ord = 1)

def fit(self, X, y, w_init = None, delta = 1e-4):
        """
                Trains the Lasso model
        Parameters
        ----------
        X : np.array of shape (n,d)
                Features
        y : np.array of shape (n,)
                Labels
        w_init : np.array of shape (d,)
                Initial guess for w
        delta : float
                Stopping criterion

        Returns
        -------
        convergence_history : array
                convergence history
        """
        iter_count = 0
        n, d = X.shape
        if w_init is None:
                w_init = np.zeros(d)
        w_prev = w_init + np.inf #just to enter the loop
        self.w = w_init
        convergence_history = []
        a = 2*np.sum(X**2, axis = 0) #precompute a
        print('shape a:', a.shape, 'should be ', d)
        while np.linalg.norm(self.w-w_prev, ord = np.inf) >= delta:
                iter_count += 1
                w_prev = np.copy(self.w)
                self.b = np.mean(y - X.dot(self.w))
                for k in range(d):
                        not_k_cols = np.arange(d) != k
                        a_k = a[k]
                        c_k = 2*np.sum(X[:,k]*(y - (self.b +
                                X[:, not_k_cols].dot(self.w[not_k_cols]))), axis = 0)
                        self.w[k] = np.float(np.piecewise(c_k,
                                [c_k < -self.reg_lambda, c_k > self.reg_lambda, ],
                                [(c_k+self.reg_lambda)/a_k, (c_k-self.reg_lambda)/a_k, 0]))
                if iter_count % 1 == 0:
                        print('Iter ', iter_count, ' Loss:', self.objective(X,y))
                convergence_history.append(self.objective(X,y))
        self.conv_history = convergence_history
        print('converged in: ', len(convergence_history))
        return convergence_history


def predict(self, X):
        """
```

```
                Use the trained model to predict values for each instance in X
                Arguments:
                        X is a n-by-d numpy array
                Returns:
                        an n-by-1 numpy array of the predictions
                """
                return X.dot(self.w) + self.b

def mse(x, y):
    return np.mean((x-y)**2)

if __name__ == "__main__":
        df_train = pd.read_table('data/crime-train.txt')
        df_test = pd.read_table('data/crime-test.txt')

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

        nonzeros = []
        w_regularization_path = []
        train_mse = []
        test_mse = []
        lambda_max = np.max(np.sum(2*X_train.values*(y_train.values -
                np.mean(y_train.values))[:, None], axis = 0))
        lambdas = [lambda_max/(2**i) for i in range(17)]

        w_init = None
        for reg_lambda in lambdas:
            model = Lasso(reg_lambda = reg_lambda)
            model.fit(X_train.values,y_train.values, w_init, delta = 1e-4)
            w_init = np.copy(model.w) #initialize with the previous solution, this is even faster and the
            w_regularization_path.append(np.copy(model.w))
            total_num_of_nonzeros = np.sum(abs(model.w) > 1e-14)
            nonzeros.append(total_num_of_nonzeros)
            train_mse.append(mse(model.predict(X_train), y_train))
            test_mse.append(mse(model.predict(X_test), y_test))
            print('Current nonzero number:', np.sum(abs(model.w) > 1e-14))

        #Part a
        plt.figure(figsize = (15,10))
        plt.plot(lambdas, nonzeros, '-o')
        plt.xscale('log')
        plt.title('A5.a: Nonzero count vs lambda')
        plt.xlabel('lambda')
        plt.ylabel('# of nonzero elements of w')
        plt.savefig('figures/A5a.pdf')
        plt.show()

        #Part b
        plt.figure(figsize = (15,10))
        w_regularization_path = np.array(w_regularization_path)
        coeffs_names = ['agePct12t29', 'pctWSocSec', 'pctUrban', 'agePct65up', 'householdsize']
        coeffs_indices = [X_train.columns.get_loc(i) for i in coeffs_names]
```

```python
for coeff_path, label in zip(w_regularization_path[:, coeffs_indices].T, coeffs_names):
    plt.plot(lambdas, coeff_path, '-o', label=label,)
plt.legend()
plt.xscale('log')
plt.title('A5.b: Regularization paths')
plt.xlabel('lambda')
plt.ylabel('Coefficient')
plt.savefig('figures/A5b.pdf')

#Part c
plt.figure(figsize = (15,10))
plt.plot(lambdas, train_mse, '-o', label = 'train_mse')
plt.plot(lambdas, test_mse, '-o', label = 'test_mse')
plt.xscale('log')
plt.title('A5.c: MSE')
plt.legend()
plt.xlabel('lambda')
plt.ylabel('error')
plt.savefig('figures/A5c.pdf')
plt.show()

#Part d
model = Lasso(reg_lambda = 30)
model.fit(X_train.values,y_train.values, w_init, delta = 1e-4)
plt.figure(figsize = (15,10))
plt.plot(model.w, '-o')
plt.title('A5.d: Weights')
plt.xlabel('Feature index')
plt.ylabel('Weight')
plt.savefig('figures/A5d.pdf')
plt.show()

print('Largest positive weight:', X_train.columns[np.argmax(model.w)])
print('Largest negative weight:', X_train.columns[np.argmin(model.w)])
```

Code for A5

A.6 **Solution:**

a. Note that $\exp(-y_i(b + x_i^T w)) = \frac{1}{\mu_i(w,b)} - 1$. Then, $\nabla_w J(w,b) = \frac{1}{n}\sum_i \nabla_w \log(1 + \exp(-y_i(b + x_i^T w))) + \nabla_w \lambda\|w\|^2 = \frac{1}{n}\sum_i \mu_i(w,b)(\frac{1}{\mu_i(w,b)} - 1)(-y_i)x_i + 2\lambda w$. So

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

Now, $\nabla_b J(w,b) = \frac{1}{n}\sum_i \nabla_b \log(1 + \exp(-y_i(b + x_i^T w))) + \nabla_b \lambda\|w\|^2 = \frac{1}{n}\sum_i \mu_i(w,b)(\frac{1}{\mu_i(w,b)} - 1)(-y_i)$. So

$$\boxed{\nabla_b J(w,b) = \frac{1}{n}\sum_i (\mu_i(w,b) - 1)y_i}$$

b. See Figure 4.

c. See Figure 5.

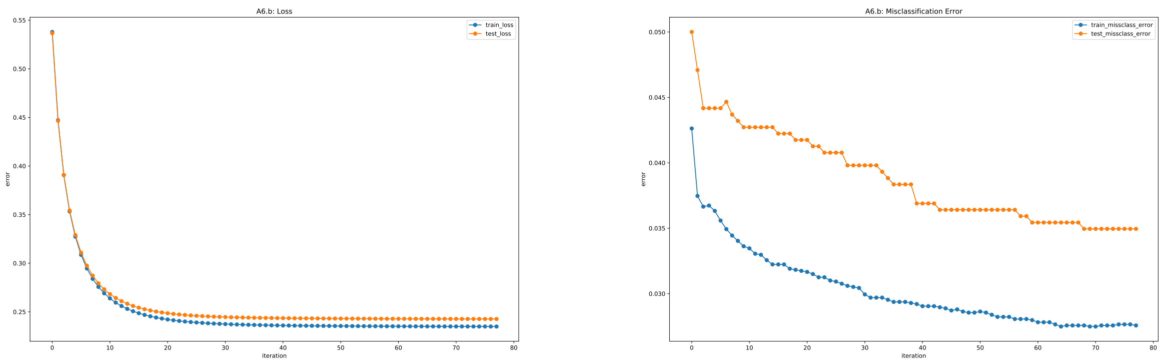

Figure 4: Problem A6.b Left: A6.bi, Right: A6.bii (Plots for Gradient Descent)

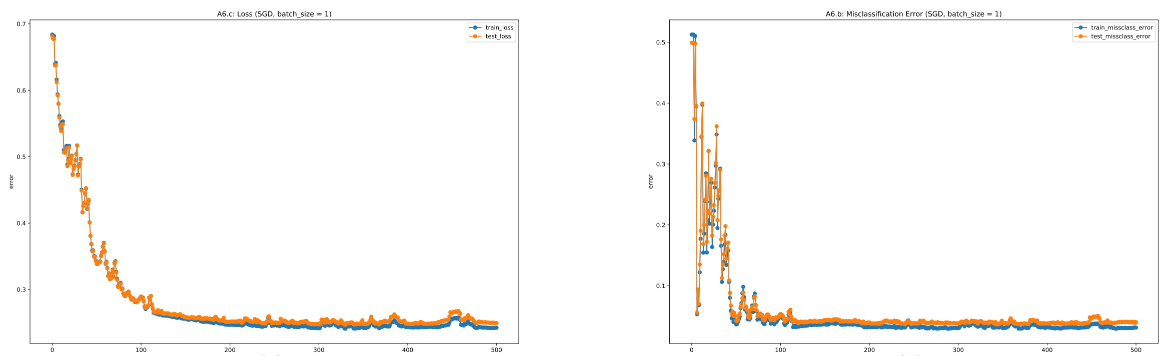

Figure 5: Problem A6.c Left: A6.ci, Right: A6.cii (Plots for Stochastic Gradient Descent with batch 1.)

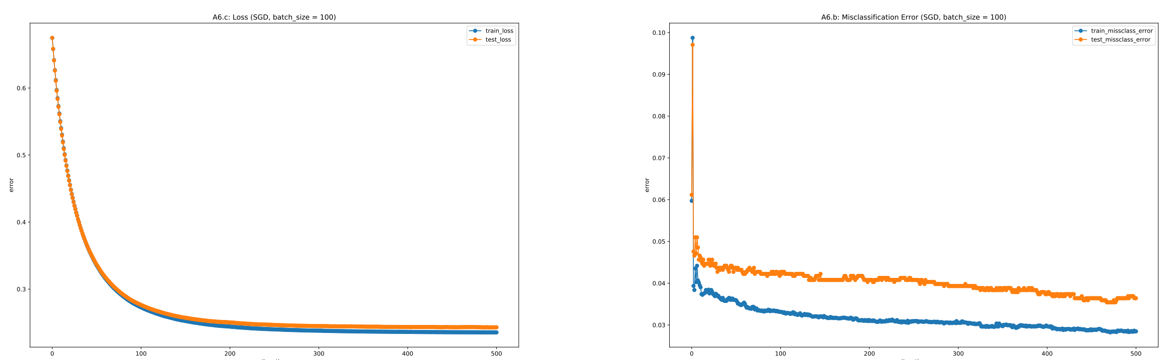

Figure 6: Problem A6.d Left: A6.di, Right: A6.dii (Plots for Stochastic Gradient Descent with batch 100.)

d. See Figure 6.

```
import numpy as np
from mnist import MNIST
import matplotlib.pyplot as plt
```

```python
def load_dataset():
    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
    return X_train, labels_train, X_test, labels_test


def accuracy_error(y_true, y_pred):
    """
    Misclass error
    Parameters
    ----------
    y_true : np.array of shape (m,)
        Vector of true labels
    y_pred : np.array of shape (m,)
        Vector of predicted labels

    Returns
    -------
    float
        error: 1-accuracy
    """
    return 1-np.mean(y_true == y_pred)

def mu(w, b, X, y):
    return 1/(1+np.exp(-y*(b + X.dot(w))))

def grad_w(w, b, X, y, reg_lambda):
    return np.mean(((mu(w, b, X, y) - 1)*y)[:,None]*X, axis = 0) +2*reg_lambda*w

def grad_b(w, b, X, y):
    return np.mean((mu(w, b, X, y) - 1)*y, axis = 0)

def J(w, b, X, y, reg_lambda = 0.1):
    return np.mean(np.log(1+np.exp(-y*(b + X.dot(w))))) + reg_lambda*w.dot(w)

def grad_descent(step, X, y, reg_lambda = 0.1, w_init = None, b_init = None, max_iter = 10000):
    n, d = X.shape
    if w_init is None:
        w_init = np.zeros(d)
    if b_init is None:
        b_init = 0
    count = 0
    w = w_init
    b = b_init
    w_prev = w_init + np.inf
    conv_history = []
    w_history = []
    b_history = []
    while np.linalg.norm(w - w_prev, np.inf) >= 1e-4 and count <= max_iter:
        count += 1
        w_prev = np.copy(w)
```

```
        w = w - step*grad_w(w, b, X, y, reg_lambda)
        b = b - step*grad_b(w, b, X, y)
        conv_history.append(J(w, b, X, y, reg_lambda))
        w_history.append(w)
        b_history.append(b)
        if count%10 == 0:
            print('Iter ', count, 'Loss: ', conv_history[-1])
    return w, b, conv_history, w_history, b_history

def predict(w, b, X):
    return np.sign(b + X.dot(w))

def SGD(step, batch_size, X, y, reg_lambda = 0.1, w_init = None, b_init = None, max_iter = 10000):
    n, d = X.shape
    if w_init is None:
        w_init = np.zeros(d)
    if b_init is None:
        b_init = 0
    count = 0
    w = w_init
    b = b_init
    w_prev = w_init + np.inf
    conv_history = []
    w_history = []
    b_history = []
    while np.linalg.norm(w - w_prev, np.inf) >= 1e-4 and count <= max_iter:
        #Sample random batch:
        batch_idx = np.random.choice(n, batch_size)
        X_batch = X[batch_idx]
        y_batch = y[batch_idx]

        count += 1
        w_prev = np.copy(w)
        w = w - step*grad_w(w, b, X_batch, y_batch, reg_lambda)
        b = b - step*grad_b(w, b, X_batch, y_batch)
        conv_history.append(J(w, b, X, y, reg_lambda))
        w_history.append(w)
        b_history.append(b)
        if count%10 == 0:
            print('Iter ', count, 'Loss: ', conv_history[-1])
    return w, b, conv_history, w_history, b_history

if __name__ == "__main__":
        X_train_mult, labels_train_mult, X_test_mult, labels_test_mult = load_dataset()
        #take only binary for 2 and 7
        idx_2_7 = (labels_train_mult == 2).astype('int') + (labels_train_mult == 7).astype('int')
        X_train = X_train_mult[idx_2_7.astype('bool')].astype('float')
        y_train = labels_train_mult[idx_2_7.astype('bool')].astype('float')
        y_train[y_train == 7] = 1
        y_train[y_train == 2] = -1
        idx_2_7_test = (labels_test_mult == 2).astype('int') + (labels_test_mult == 7).astype('int')
        X_test = X_test_mult[idx_2_7_test.astype('bool')].astype('float')
        y_test = labels_test_mult[idx_2_7_test.astype('bool')].astype('float')
        y_test[y_test == 7] = 1
        y_test[y_test == 2] = -1
```

```
    w, b, conv_history, w_history, b_history = grad_descent(
0.1, X_train, y_train, reg_lambda = 0.1, w_init = None, b_init = None, max_iter = 10000)
    #Part b1
    train_loss = conv_history
    test_loss = [J(w, b, X_test, y_test, reg_lambda = 0.1) for w, b in zip(w_history, b_history)]

    plt.figure(figsize = (15,10))
    plt.plot(train_loss, '-o', label = 'train_loss')
    plt.plot(test_loss, '-o', label = 'test_loss')
    plt.title('A6.b: Loss')
    plt.legend()
    plt.xlabel('iteration')
    plt.ylabel('error')
    plt.savefig('figures/A6b1.pdf')
    plt.show()

    #Part b2
    y_pred_train = [predict(w, b, X_train) for w, b in zip(w_history, b_history)]
    train_missclass_error = [accuracy_error(y_train, y_pred) for y_pred in y_pred_train]

    y_pred_test = [predict(w, b, X_test) for w, b in zip(w_history, b_history)]
    test_missclass_error = [accuracy_error(y_test, y_pred) for y_pred in y_pred_test]

    plt.figure(figsize = (15,10))
    plt.plot(train_missclass_error, '-o', label = 'train_missclass_error')
    plt.plot(test_missclass_error, '-o', label = 'test_missclass_error')
    plt.title('A6.b: Misclassification Error')
    plt.legend()
    plt.xlabel('iteration')
    plt.ylabel('error')
    plt.savefig('figures/A6b2.pdf')
    plt.show()

    w, b, conv_history, w_history, b_history = SGD(
            step = 0.01, batch_size = 1, X = X_train, y = y_train, max_iter = 500)
    #Part c1
    train_loss = conv_history
    test_loss = [J(w, b, X_test, y_test, reg_lambda = 0.1) for w, b in zip(w_history, b_history)]

    plt.figure(figsize = (15,10))
    plt.plot(train_loss, '-o', label = 'train_loss')
    plt.plot(test_loss, '-o', label = 'test_loss')
    plt.title('A6.c: Loss (SGD, batch_size = 1)')
    plt.legend()
    plt.xlabel('iteration')
    plt.ylabel('error')
    plt.savefig('figures/A6c1.pdf')
    plt.show()

    #Part c2
    y_pred_train = [predict(w, b, X_train) for w, b in zip(w_history, b_history)]
    train_missclass_error = [accuracy_error(y_train, y_pred) for y_pred in y_pred_train]

    y_pred_test = [predict(w, b, X_test) for w, b in zip(w_history, b_history)]
```

```
    test_missclass_error = [accuracy_error(y_test, y_pred) for y_pred in y_pred_test]

    plt.figure(figsize = (15,10))
    plt.plot(train_missclass_error, '-o', label = 'train_missclass_error')
    plt.plot(test_missclass_error, '-o', label = 'test_missclass_error')
    plt.title('A6.b: Misclassification Error (SGD, batch_size = 1)')
    plt.legend()
    plt.xlabel('iteration')
    plt.ylabel('error')
    plt.savefig('figures/A6c2.pdf')
    plt.show()

    w, b, conv_history, w_history, b_history = SGD(
step = 0.01, batch_size = 100, X = X_train, y = y_train, max_iter = 500)
    #Part d1
    train_loss = conv_history
    test_loss = [J(w, b, X_test, y_test, reg_lambda = 0.1) for w, b in zip(w_history, b_history)]

    plt.figure(figsize = (15,10))
    plt.plot(train_loss, '-o', label = 'train_loss')
    plt.plot(test_loss, '-o', label = 'test_loss')
    plt.title('A6.c: Loss (SGD, batch_size = 100)')
    plt.legend()
    plt.xlabel('iteration')
    plt.ylabel('error')
    plt.savefig('figures/A6d1.pdf')
    plt.show()

    #Part c2
    y_pred_train = [predict(w, b, X_train) for w, b in zip(w_history, b_history)]
    train_missclass_error = [accuracy_error(y_train, y_pred) for y_pred in y_pred_train]

    y_pred_test = [predict(w, b, X_test) for w, b in zip(w_history, b_history)]
    test_missclass_error = [accuracy_error(y_test, y_pred) for y_pred in y_pred_test]

    plt.figure(figsize = (15,10))
    plt.plot(train_missclass_error, '-o', label = 'train_missclass_error')
    plt.plot(test_missclass_error, '-o', label = 'test_missclass_error')
    plt.title('A6.b: Misclassification Error (SGD, batch_size = 100)')
    plt.legend()
    plt.xlabel('iteration')
    plt.ylabel('error')
    plt.savefig('figures/A6d2.pdf')
    plt.show()
```

Code for A6

B.4 **Solution:**

a. No time to explain!:)

b. No time to explain!:)

c. For Multinomial Logistic Regression trained with L(W) the train accuracy is 0.9127 and test accuracy is 0.9158. For J(W) these values are 0.8490 and 0.8537 respectively. Both models were trained for 50 epochs with learning rate $l = 0.01$.

```
import torch
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from tqdm import tqdm
import torch.nn as nn

to_tensor = transforms.ToTensor()

mnist_trainset = datasets.MNIST(root='./data', train=True, download=True, transform=to_tensor)
mnist_testset = datasets.MNIST(root='./data', train=False, download=True, transform=to_tensor)
train_loader = torch.utils.data.DataLoader(mnist_trainset,
                                           batch_size=128,
                                           shuffle=True)
test_loader = torch.utils.data.DataLoader(mnist_testset,
                                          batch_size=128,
                                          shuffle=True)


def train_CrossEntropy(l, num_epochs, train_loader):

    # initialize W
    W = torch.zeros(784, 10, requires_grad = True)
    # define loss function
    criterion = nn.CrossEntropyLoss()
    for epoch in range(num_epochs):
        # iterate through batches
        for inputs, labels in tqdm(iter(train_loader)):
            # flatten images
            inputs = torch.flatten(inputs, start_dim=1, end_dim=3)
            # compute predictions
            preds = torch.matmul(inputs, W)
            # compute loss
            loss = torch.nn.functional.cross_entropy(preds, labels)
            # computes derivatives of the loss with respect to W
            loss.backward()
            # gradient descent update
            W.data = W.data - l * W.grad
            W.grad.zero_()
        print("Loss\t{}".format(loss))
    return W

def train_MSE(l, num_epochs, train_loader):

    # initialize W
    W = 0.001*torch.rand(784, 10, requires_grad = True)
    for epoch in range(num_epochs):
        # iterate through batches
        for inputs, labels in tqdm(iter(train_loader)):
            # flatten images
            inputs = torch.flatten(inputs, start_dim=1, end_dim=3)
            # compute predictions
            preds = torch.matmul(inputs, W)
            # convert labels to one-hot labels
            y_onehot = torch.zeros(preds.shape[0], preds.shape[1])
```

```
            y_onehot.scatter_(1, labels.unsqueeze(1), 1)
            # compute loss
            loss = torch.mean(torch.norm((preds-y_onehot), dim = 1)**2)/2
            # computes derivatives of the loss with respect to W
            grad = torch.autograd.grad(loss, [W])
            # gradient descent update
            W.data = W.data - l * grad[0]
        print("Loss\t{}".format(loss))
    return W



def compute_accuracy(W, data_loader):
    acc = 0
    for inputs, labels in tqdm(iter(test_loader)):
        inputs = torch.flatten(inputs, start_dim=1, end_dim=3)
        preds = torch.argmax(torch.matmul(inputs, W),1)
        acc += torch.sum(preds == labels)

    acc = acc.to(dtype=torch.float)/len(test_loader.dataset)
    return(acc)


if __name__ == '__main__':


    W_CE = train_CrossEntropy(0.001, 50, train_loader)
    W_MSE = train_MSE(0.001, 50, train_loader)

    acc_CE_test = compute_test_accuracy(W_CE, test_loader)
    acc_MSE_test = compute_test_accuracy(W_MSE, test_loader)
    acc_CE_train = compute_test_accuracy(W_CE, train_loader)
    acc_MSE_train = compute_test_accuracy(W_MSE, train_loader)
    print('CE test accuracy: ', acc_CE_test)
    print('MSE test accuracy: ', acc_MSE_test)
    print('CE train accuracy: ', acc_CE_train)
    print('MSE train accuracy: ', acc_MSE_train)
```

Code for B4