Homework #2

CSE 446: Machine Learning

Eric Boris: 1976637

Conceptual Questions

Problem 0

a. **No**, because it's possible that there are other highly correlated features that also have large positive weights. In which case, any one could be removed without affecting the quality of the predictions.

b. The L1 norm penalty results in a more sparse features than the L2 norm because the **L1 ball is diamond shaped with vertices on the axes** while the **L2 norm is spherical**. This means that an intersection between the function's contour lines with the L1 ball are more likely to occur on an axis, resulting in zeros for the other axes, than with the L2 norm.

c. Advantage: Better **weight sparsity** than L1. Disadvantage: **Non-Convex**.

d. k-fold cross validation works by dividing the training data into k subsets. For k iterations, one of the k subsets is used as a test set and the model is trained over the remaining k-1 subsets. The resulting error is found by averaging the error over the k iterations. **Because the training set using k-fold cross is smaller**, k-1 than for example n-1 as in LOOCV, **the bias is higher**. But training time is reduced for k-fold cross from LOOCV. In short, increasing k increases the size of the training set and therefore, increases training time and reduces bias. **k=10** could be a reasonable choice because k is sufficiently large to have **low bias but fast training times**.

e. **True** because the gradient update values can get stuck stepping over the minimum.

f. Because **on average it goes in the direction of the true gradient**.

g. Advantage: SGD is **less computationally intensive** than GD. Disadvantage: SGD requires **more training** than GD to achieve the same error.

Convexity and Norms

Problem 1

a. Prove that $\|x\| = f(x) = \left(\sum_{i=1}^{n} x_i^2\right)^{\frac{1}{2}}$ is a norm.

- Non-negativity: $\|x\| \geq 0$ for all $x_i \in \mathbb{R}^n$ since $x_i^2 \geq 0$ and $\|x\| = 0$ if and only if $x = 0$.

- Absolute scalability: $\|ax\| = |a|\|x\|$ for all $a \in \mathbb{R}$ and $x \in \mathbb{R}^n$.

$$\begin{aligned}
\|ax\| &= \left(\sum_{i=1}^{n}(ax_i)^2\right)^{\frac{1}{2}} \\
&= \left(\sum_{i=1}^{n} a^2 x_i^2\right)^{\frac{1}{2}} \\
&= \left(a^2 \sum_{i=1}^{n} x_i^2\right)^{\frac{1}{2}} \\
&= \left(a^2\right)^{\frac{1}{2}} \left(\sum_{i=1}^{n} x_i^2\right)^{\frac{1}{2}} \\
&= |a| \left(\sum_{i=1}^{n} x_i^2\right)^{\frac{1}{2}} && \text{since } \left(a^2\right)^{\frac{1}{2}} > 0 \text{ for all } a \in \mathbb{R} \setminus 0 \\
&= |a|\|x\|
\end{aligned}$$

- Triangle inequality: $\|x + y\| \leq \|x\| + \|y\|$ for all x, $y \in \mathbb{R}^n$. Let $\langle x, y\rangle$ be the inner product $x \cdot y = \sum_{i=1}^{n} x_i y_i$. Prove using Cauchy-Schwarz inequality, i.e. $|\langle x, y\rangle|^2 \leq \langle x, x\rangle \cdot \langle y, y\rangle$.

$$\begin{aligned}
(\|x + y\|)^2 &= \|x\|^2 + 2\langle x, y\rangle + \|y\|^2 \\
&\leq \|x\|^2 + 2\left(\|x\| \cdot \|y\|\right) + \|y\|^2 \\
&= (\|x\| + \|y\|)^2
\end{aligned}$$

Thus $\|x + y\| \leq \|x\| + \|y\|$ since $(\|x + y\|)^2 = (\|x\| + \|y\|)^2$ and all are positive by non-negativity.

b. Prove by contradiction that $g(x) = \left(\sum_{i=1}^{n}|x_i|^{\frac{1}{3}}\right)^3$ is not a norm.

Assume that $g(x)$ is a norm.
By the triangle inequality $g(x + y) = g(x) + g(y)$ for all $x, y \in \mathbb{R}^n$.
Let $x = [1, 0]^T$ and $y = [0, 1]^T$.
Trivially, $x, y \in \mathbb{R}^n$.
$g(x + y) = (1 + 1)^3 = 8$
$g(x) + g(y) = 1^3 + 1^3 = 2$
$g(x + y) \neq g(x) + g(y)$

Thus, since $g(x)$ is a norm and the triangle inequality doesn't hold is a contradiction, $g(x)$ is not a norm.

Problem 2

- **I is convex**.
- **II is not convex** because points b and c are in the set but the line segment $\overline{\text{bc}}$ is not.
- **III is not convex** because points a and d are in the set but the line segment $\overline{\text{ad}}$ is not.

Problem 3

- **I is convex** on $[a, c]$.
- **II is not convex** on $[a, d]$ because $f(\lambda c + (1 - \lambda)d) \geq \lambda f(c) + (1 - \lambda)f(d)$ on $[c, d]$ for an arbitrary λ.

- **II is convex** on $[a, b]$.
- **III is not convex** on $[a, c]$ because $f(\lambda a + (1-\lambda)b) \geq \lambda f(a) + (1-\lambda)f(b)$ on $[a, b]$ and $f(\lambda b + (1-\lambda)c) \geq \lambda f(b) + (1-\lambda)f(c)$ on $[b, c]$ for an arbitrary λ.

Problem 4

Prove that $f(x) = (x^T Ax)^{\frac{1}{2}}$ is convex given that $A \in \mathbb{R}^{n \times n}$ is a symmetric positive definite matrix.

Prove that $f(x)$ is a norm. Since A is a real symmetric matrix, A is orthogonally diagonizable. By spectral decomposition, there exist matrices $V, \Lambda \in \mathbb{R}^{n \times n}$ such that V is orthogonal and $\Lambda = \text{diag}(\lambda_1, ..., \lambda_n)$ such that $A = V\Lambda V^T$. Because A is positive definite, for all $i \in [1, ..., n]$, $\lambda_i > 0$. Then $\Lambda^{\frac{1}{2}} = \text{diag}(\lambda_1^{\frac{1}{2}}, ..., \lambda_n^{\frac{1}{2}})$ holds. So we can write $f(x) = \|\Lambda V^T x\|_2$. Since ΛV^T is invertible, $\lambda_n \|x\|_2^2 \leq \langle x, Ax \rangle \leq \lambda_1 \|x\|_2^2$. Which implies that $\sqrt{\lambda_n}\|x\|_2^2 \leq \|x\|_A \leq \sqrt{\lambda_1}\|x\|_2^2$. Hence, $f(x) = \|\Lambda^{\frac{1}{2}} V^T x\|_2$. Thus $f(x)$ is a norm. Prove that all norms are convex. By the definition of convexity, an arbitrary function $g(x)$ is convex if and only if sub-additivity holds. Since $g(x)$ is a norm, the triangle inequality $\|\lambda x + (1-\lambda)y\| \leq \lambda\|x\| + (1-\lambda)\|y\|$ holds for all $x, y \in \mathbb{R}^n$ and $\lambda \in [1, 0]$ by problem 1a. By homogeneity then, $g(\lambda x + (1-\lambda)y) = \lambda g(x) + (1-\lambda)g(y)$. Therefore, all norms are convex. And since $f(x)$ is a norm, $f(x)$ is convex.

Lasso on a real dataset

Problem 5

a. See Figure 1.

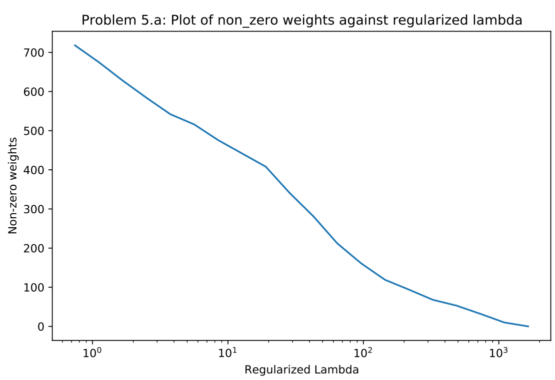

Figure 1

b. See Figure 2.

c. We see that higher λ values increase the number of zeros in the weights leading to a more sparse solution. But we encounter problems when λ is too small or too large. When λ is too small, we have a hight false discovery rate and when λ is too large the weights become too sparse and the model has poor performance. These data suggest that we should look for values of lambda that balance these trade-offs.

```
# Lasso Part 1 - Problem 5

import sys
import numpy as np
```

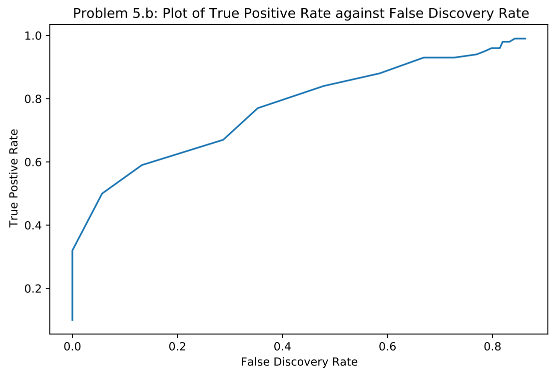

Figure 2

```python
import matplotlib.pyplot as plt
from Lasso import Lasso
import Performance as pf
import Supplemental as sp
import DataManagement as dm

def part_a(lambdas, non_zeros):
    plt.figure(figsize=(8, 5))
    plt.plot(lambdas, non_zeros)
    plt.title('Problem 5.a: Plot of non_zero weights against regularized lambda')
    plt.xscale('log')
    plt.ylabel('Non-zero weights')
    plt.xlabel('Regularized Lambda')
    plt.savefig('P5_a.pdf')
    plt.show()

def part_b(FDR, TPR):
    plt.figure(figsize=(8, 5))
    plt.plot(FDR, TPR)
    plt.title('Problem 5.b: Plot of True Positive Rate against False Discovery Rate')
    plt.ylabel('True Postive Rate')
    plt.xlabel('False Discovery Rate')
    plt.savefig('P5_b.pdf')
    plt.show()

def main(args):
    # Training data values.
    if len(args) == 5:
        n = args[1]
        d = args[2]
        k = args[3]
        sigma = args[4]
    else:
        n = 500
        d = 1000
        k = 100
        sigma = 1

    # Get the synthetic data.
    X, y, w_actual = dm.synthetic_data(n, d, k, sigma)

    # False Discovery Rate (FDR) and True Positive Rate (TPR).
    FDR = []
    TPR = []

    # Count of nonzero weights.
    non_zeros = []

    # Let begin as the max lambda value
```

```
    # and be reduced by constant ratio 1.5.
    max_lam = sp.max_lambda(X, y)

    # Let lambdas hold the precomputed regularized lambdas.
    regularized_lambdas = sp.regularized_lambdas(max_lam)

    # Passing in the values of w_pred into the model is faster
    # than initializing with 0 weights each time.
    w_pred = None

    for rl in regularized_lambdas:
        print(f'Lambda: {rl}')

        # Train the model.
        model = Lasso(rl)
        model.train(X, y, w_pred, delta=1E-3, verbose=True)
        w_pred = model.w

        # Count non-zeros for part a.
        non_zeros.append(np.sum(abs(w_pred) > 1E-14))

        FDR.append(pf.fdr(w_actual, w_pred))
        TPR.append(pf.tpr(w_actual, w_pred))

    # Plot the graphs for part a and part b.
    part_a(regularized_lambdas, non_zeros)
    part_b(FDR, TPR)

if __name__ == '__main__':
    main(sys.argv)
```

Problem 6

a.
- Percentage of population that is of hispanic heritage (racePctHisp): Hispanic is a race term with a meaning that has historically been changed as a result of policy decisions.
- Number of people living in areas classified as urban (numbUrban): Urban is a term with a meaning that has changed as a result of policy decisions.
- Percentage of people under the poverty level (PctPopUnderPov): Poverty level has changed as a result of policy decisions.

b.
- Percentage of males who are divorced (MalePctDivorce): It might be assumed that higher rates of divorce cause higher crime rates but it could also be the case that higher crime rates case in an area cause higher divorce rates.
- Number of vacant households (HousVacant): It might be assumed that more vacant housholds cause higher crime rates but it could also be the case that higher crime rates cause there to be more vacant houses.
- Number of homeless people counted in the street (NumStreet): It might be assumed that more homeless people in the street cause higher crime rates but it could also be the case that higher crime rates cause there to be more homeless people in the street.

a. See Figure 3.

b. See Figure 4.

c. See Figure 5.

d. See Figure 6.
Minimum weight: index 39, PctKids2Par with value -0.1247.
Maximum weight: index 45, PctIlleg with value 0.0687.

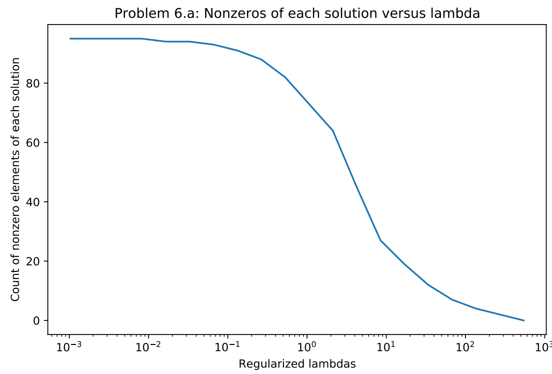

Figure 3

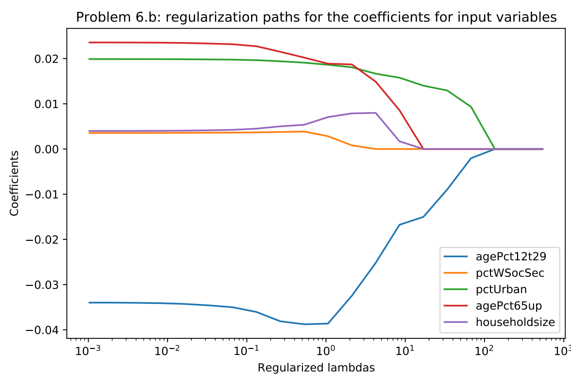

Figure 4

Figure 5

Crime rates correlate most positively with "percentage of kids born to never married" (PctIlleg) and correlate most negatively with "percentage of kids in family housing with 2 parents" (PctKids2Par). These data suggest that crime rates increase with increases in percentage of kids born to never married and decrease with increases in percentage of kids in family housing with 2 parents.

e. The dictum that "Correlation does not equal Causation" holds. More over-65 years old people living in an area doesn't cause lower crime rates. There are many confounding factors. One could be that older people move away from high crime rate areas. So rather than there presence indicating low crime rates

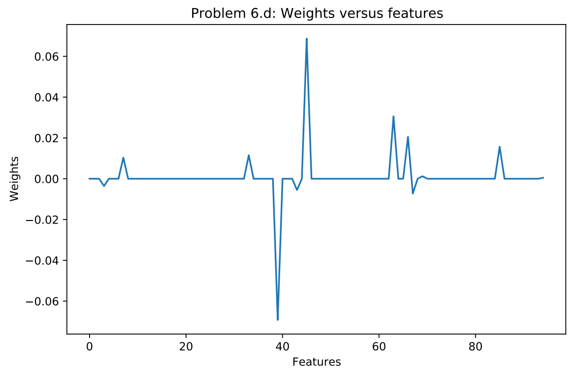

Figure 6

their absence can indicate high crime rates. And, should older people return, the crime rates could remain the same, or even increase, if we assume that older people are more likely to be targeted by criminals than younger people.

```
# Lasso Part 2 - Problem 6

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from Lasso import Lasso
import DataManagement as dm
import Supplemental as sp
import Performance as pf

def part_a(regularized_lambdas, non_zeros):
  ''' Plot the graph defined in part a. '''
  plt.figure(figsize=(8, 5))
  plt.plot(regularized_lambdas, non_zeros)
  plt.title('Problem 6.a: Nonzeros of each solution versus lambda')
  plt.xlabel('Regularized lambdas')
  plt.ylabel('Count of nonzero elements of each solution')
  plt.xscale('log')
  plt.savefig('P6_a.pdf')
  plt.show()

def part_b(regularization_paths, regularized_lambdas, coefficient_names, coefficient_indices):
  ''' Plot the graph defined in part b. '''
  plt.figure(figsize=(8, 5))
  for coeff, label in zip(np.array(regularization_paths)[:, coefficient_indices].T,
      coefficient_names):
    plt.plot(regularized_lambdas, coeff, label=label)
  plt.title('Problem 6.b: regularization paths for the coefficients for input variables')
  plt.xlabel('Regularized lambdas')
  plt.ylabel('Coefficients')
  plt.xscale('log')
  plt.legend()
  plt.savefig('P6_b.pdf')
  plt.show()

def part_c(regularized_lambdas, train_mse, test_mse):
  ''' Plot the graph defined in part c. '''
  plt.figure(figsize=(8, 5))
  plt.plot(regularized_lambdas, train_mse, label='train_mse')
  plt.plot(regularized_lambdas, test_mse, label='test_mse')
  plt.title('Problem 6.c: Squared error on the training and test data versus lambda')
  plt.xlabel('Regularized lambdas')
  plt.ylabel('Training and test squared error')
  plt.xscale('log')
```

```python
    plt.savefig('P6_c.pdf')
    plt.show()

def part_d(weights):
    ''' Plot the graph defined in pard d. '''
    plt.figure(figsize=(8, 5))
    plt.plot(weights)
    plt.title('Problem 6.d: Weights versus features')
    plt.xlabel('Features')
    plt.ylabel('Weights')
    plt.savefig('P6_d.pdf')
    plt.show()

def main():
    # Load the data frames.
    df_train = pd.read_table('data/crime-train.txt')
    df_test = pd.read_table('data/crime-test.txt')

    # Split the data and labels.
    # Let column be the column to split the data frames on.
    column = 'ViolentCrimesPerPop'
    X_train, y_train = dm.split_on_column(df_train, column)
    X_test, y_test = dm.split_on_column(df_test, column)

    # Let the following be the list of regularized lambdas to train over.
    max_lambda = sp.max_lambda(X_train, y_train)
    regularized_lambdas = sp.regularized_lambdas(max_lambda, n=20, constant=2)

    # Let the following hold data relevant to each training iteration for plotting graphs.
    non_zeros = []
    train_mse = []
    test_mse = []
    paths = []

    # Passing in the values of w_pred into the model is faster
    # than initializing with 0 weights each time.
    w_pred = None

    for rl in regularized_lambdas:
        print(f'Lambda: {rl}')

        # Train the model.
        model = Lasso(rl)
        model.train(X_train.values, y_train.values, w_pred, delta=1E-4, verbose=True)

        # Use as the initialization weights on the next iteration.
        # Must copy to prevent side effects.
        w_pred = np.copy(model.w)
        paths.append(w_pred)

        # Run the model.
        y_hat_train = model.predict(X_train)
        y_hat_test = model.predict(X_test)

        # Record model performance.
        non_zeros.append(pf.non_zeros(w_pred))
        train_mse.append(pf.mse(y_train, y_hat_train))
        test_mse.append(pf.mse(y_test, y_hat_test))

    # Plot part a.
    part_a(regularized_lambdas, non_zeros)

    # Plot part b.
    coefficient_names = ['agePct12t29', 'pctWSocSec', 'pctUrban', 'agePct65up', 'householdsize']
    coefficient_indices = [X_train.columns.get_loc(name) for name in coefficient_names]
    part_b(paths, regularized_lambdas, coefficient_names, coefficient_indices)

    # Plot part c.
    part_c(regularized_lambdas, train_mse, test_mse)
```

```

  # Plot part d.
  model_d = Lasso(30)
  model_d.train(X_train.values, y_train.values, w_pred, delta=1E-4, verbose=True)
  part_d(model_d.w)

  min_w_val = float('inf')
  min_w_idx = None
  max_w_val = float('-inf')
  max_w_idx = None
  for i, w in enumerate(model_d.w):
    if w > max_w_val:
      max_w_val = w
      max_w_idx = i
    if w < min_w_val:
      min_w_val = w
      min_w_idx = i
  print(f'Min Weight: index={min_w_idx} val={min_w_val}')
  print(f'Max Weight: index={max_w_idx} val={max_w_val}')

  for i, weight in enumerate(model_d.w):
    if weight != 0:
      print(f'{i}: {weight}')

  for i, label in enumerate(X_train):
    print(f'{i}: {label}')

if __name__ == '__main__':
  main()
```

Logistic Regression

Problem 7

a. Derive the gradient $\nabla_w J(w, b)$.

$$
\begin{aligned}
J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} \log\left(1+\exp\left(-y_i\left(b+x_i^T w\right)\right)\right) + \lambda\|w\|_2^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} \log\left(1+\left(\frac{1}{\mu_i(w,b)}-1\right)\right) + \lambda\|w\|_2^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{1}{\mu_i(w,b)}\right) + \lambda\|w\|_2^2 \\
\nabla_w J(w,b) &= \nabla_w \frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{1}{\mu_i(w,b)}\right) + \nabla_w \lambda\|w\|_2^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} \mu_i(w,b)\left(\frac{1}{\mu_i(w,b)}-1\right)(-y_i)(x_i) + 2\lambda w \\
&= \frac{1}{n}\sum_{i=1}^{n} \left(\mu_i(w,b)-1\right)(y_i)x_i + 2\lambda w
\end{aligned}
$$

Thus,

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

Derive the gradient $\nabla_b J(w,b)$.

$$J(w,b) = \frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{1}{\mu_i(w,b)}\right) + \lambda\|w\|_2^2$$

$$\begin{aligned}\nabla_b J(w,b) &= \nabla_b \frac{1}{n}\sum_{i=1}^{n} \log\left(\frac{1}{\mu_i(w,b)}\right) + \nabla_b \lambda\|w\|_2^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} \mu_i(w,b)\left(\frac{1}{\mu_i(w,b)} - 1\right)(-y_i) + 0 \\ &= \frac{1}{n}\sum_{i=1}^{n} \left(\mu_i(w,b) - 1\right) y_i\end{aligned}$$

Thus,

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

.

b. • See Figure 7.

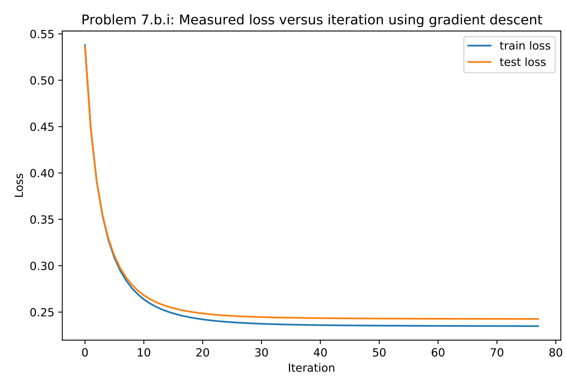

Figure 7

• See Figure 8.

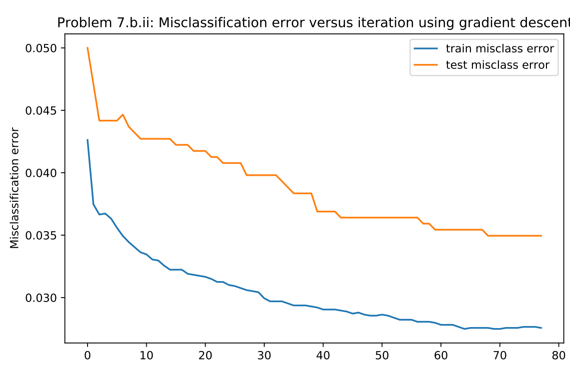

Figure 8

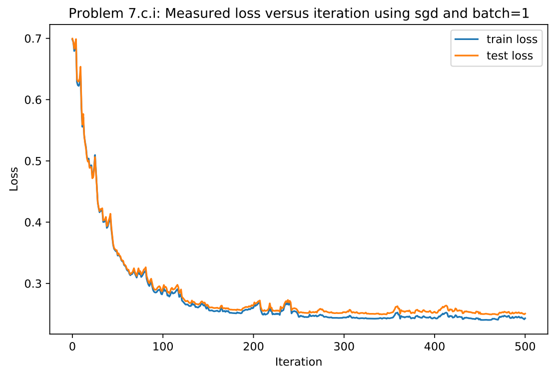

Figure 9

c. • See Figure 9.
   • See Figure 10.

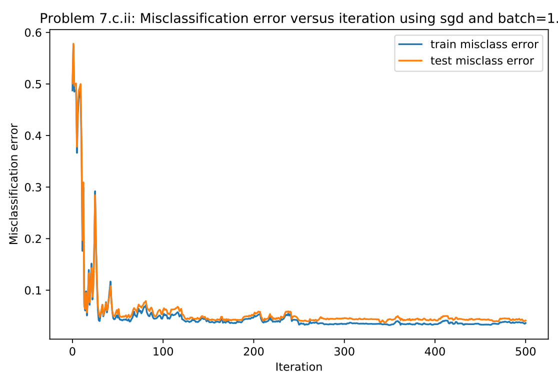

Figure 10

d. • See Figure 11.

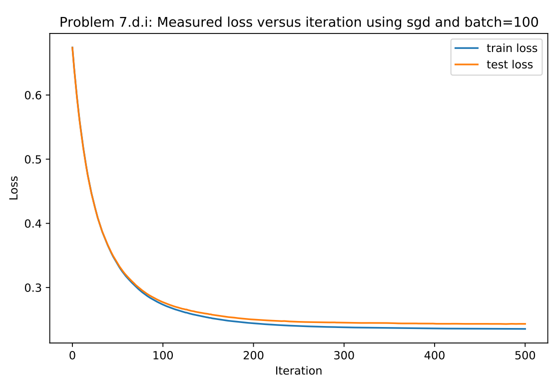

Figure 11

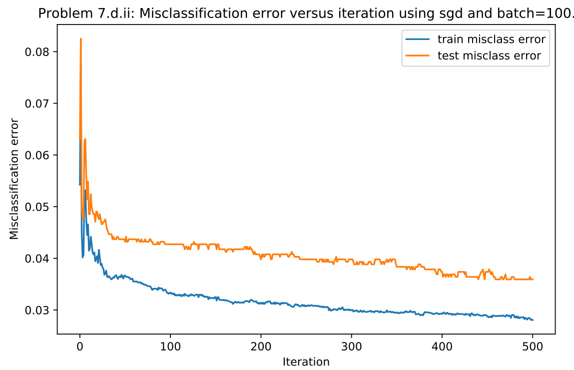

Figure 12

- See Figure 12.

```python
# Logistic Regression - Problem 7

import matplotlib.pyplot as plt
import DataManagement as dm
import Supplemental as sp
import Performance as pf
from GradientDescent import gradient_descent
from StochasticGradientDescent import stochastic_gradient_descent as sgd

def two_part_plot(x1, x1_label, x2, x2_label, title, x_label, y_label, file_name):
 plt.figure(figsize=(8, 5))
 plt.plot(x1, label=x1_label)
 plt.plot(x2, label=x2_label)
 plt.title(title)
 plt.xlabel(x_label)
 plt.ylabel(y_label)
 plt.legend()
 plt.savefig(file_name)
 plt.show()

def main():
 # Load the mnist data with all columns.
 print(f'Loading data')
 X_train_all, y_train_all, X_test_all, y_test_all = dm.load_mnist()

 # Strip from each data set all columns except columns 2 and 7.
 print(f'Stripping columns')
 keep_cols = (2, 7)

 i_train = dm.indices(y_train_all, keep_cols)
 X_train = dm.strip_cols(X_train_all, i_train)
 y_train = dm.strip_cols(y_train_all, i_train)

 i_test = dm.indices(y_test_all, keep_cols)
 X_test = dm.strip_cols(X_test_all, i_test)
 y_test = dm.strip_cols(y_test_all, i_test)

 # Set the y label values.
 print('Setting label values')
 pos, neg = 1, -1
 y_train[y_train == 7] = pos
 y_train[y_train == 2] = neg
 y_test[y_test == 7] = pos
 y_test[y_test == 2] = neg

 # Run gradient descent.
 print('Running gradient descent')
```

```python
w, b, w_history, b_history, train_loss = gradient_descent(X_train, y_train, alpha=0.1, verbose=True)

print('Plotting part b')
# Plot measured loss versus iteration.
test_loss = [sp.gradient_loss(X_test, y_test, w, b) for w, b in zip(w_history, b_history)]

#part_b_i(train_loss, test_loss)
two_part_plot(x1 = train_loss,
   x1_label = 'train loss',
   x2 = test_loss,
   x2_label = 'test loss',
   title = 'Problem 7.b.i: Measured loss versus iteration using gradient descent',
   x_label = 'Iteration',
   y_label = 'Loss',
   file_name = 'P7_b_i.pdf')

# Plot misclassification error versus iteration.
y_hat_train = [sp.gradient_predict(X_train, w, b) for w, b in zip(w_history, b_history)]
y_train_error = [pf.misclass_error(y_train, y_hat) for y_hat in y_hat_train]

y_hat_test = [sp.gradient_predict(X_test, w, b) for w, b in zip(w_history, b_history)]
y_test_error = [pf.misclass_error(y_test, y_hat) for y_hat in y_hat_test]

two_part_plot(x1 = y_train_error,
   x1_label = 'train misclass error',
   x2 = y_test_error,
   x2_label = 'test misclass error',
   title = 'Problem 7.b.ii: Misclassification error versus iteration using gradient descent',
   x_label = 'Iteration',
   y_label = 'Misclassification error',
   file_name = 'P7_b_ii.pdf')

# Run stochastic gradient descent with batch = 1.
print('Running Stochastic Gradient Descent with batch=1')
w, b, w_history, b_history, train_loss = sgd(X_train, y_train, alpha=0.01, batch_size=1, max_iterations=500, verbose=True)

print('Plotting part c')
# Plot measured loss versus iteration using sgd and batch = 1.
test_loss = [sp.gradient_loss(X_test, y_test, w, b) for w, b in zip(w_history, b_history)]

two_part_plot(x1 = train_loss,
   x1_label = 'train loss',
   x2 = test_loss,
   x2_label = 'test loss',
   title = 'Problem 7.c.i: Measured loss versus iteration using sgd and batch=1',
   x_label = 'Iteration',
   y_label = 'Loss',
   file_name = 'P7_c_i.pdf')

# Plot misclassification error versus iteration using sgd and batch = 1.
y_hat_train = [sp.gradient_predict(X_train, w, b) for w, b in zip(w_history, b_history)]
y_train_error = [pf.misclass_error(y_train, y_hat) for y_hat in y_hat_train]

y_hat_test = [sp.gradient_predict(X_test, w, b) for w, b in zip(w_history, b_history)]
y_test_error = [pf.misclass_error(y_test, y_hat) for y_hat in y_hat_test]

two_part_plot(x1 = y_train_error,
   x1_label = 'train misclass error',
   x2 = y_test_error,
   x2_label = 'test misclass error',
   title = 'Problem 7.c.ii: Misclassification error versus iteration using sgd and batch=1.',
   x_label = 'Iteration',
   y_label = 'Misclassification error',
   file_name = 'P7_c_ii.pdf')

# Run stochastic gradient descent with batch = 1.
print('Running Stochastic Gradient Descent with batch=100')
```

```
    w, b, w_history, b_history, train_loss = sgd(X_train, y_train, alpha=0.01, batch_size=100, max_iterations=500, verbose=True)

    print('Plotting part c')
    # Plot measured loss versus iteration using sgd and batch = 100.
    test_loss = [sp.gradient_loss(X_test, y_test, w, b) for w, b in zip(w_history, b_history)]

    two_part_plot(x1 = train_loss,
        x1_label = 'train loss',
        x2 = test_loss,
        x2_label = 'test loss',
        title = 'Problem 7.d.i: Measured loss versus iteration using sgd and batch=100',
        x_label = 'Iteration',
        y_label = 'Loss',
        file_name = 'P7_d_i.pdf')

    # Plot misclassification error versus iteration using sgd and batch = 100.
    y_hat_train = [sp.gradient_predict(X_train, w, b) for w, b in zip(w_history, b_history)]
    y_train_error = [pf.misclass_error(y_train, y_hat) for y_hat in y_hat_train]

    y_hat_test = [sp.gradient_predict(X_test, w, b) for w, b in zip(w_history, b_history)]
    y_test_error = [pf.misclass_error(y_test, y_hat) for y_hat in y_hat_test]

    two_part_plot(x1 = y_train_error,
        x1_label = 'train misclass error',
        x2 = y_test_error,
        x2_label = 'test misclass error',
        title = 'Problem 7.d.ii: Misclassification error versus iteration using sgd and batch=100.',
        x_label = 'Iteration',
        y_label = 'Misclassification error',
        file_name = 'P7_d_ii.pdf')

if __name__ == '__main__':
    main()
```

Additional files used for Problems 5-7

```
# Define Data Management functions for Problems 5-7.

import numpy as np
from mnist import MNIST

def synthetic_data(n, d, k, sigma):
        ''' Generate synthetic training data. '''
        # Let X be an (n, d) matrix with values drawn from a normal distribution.
        X = np.random.normal(0.0, 1.0, (n, d))

        # Let w be a (d, ) array with values d/k if d in range {1, ..., k} else 0.
        w = np.arange(d) / k
        w[k + 1: ] = 0

        # Let y be a (n, ) array with values y_i = w^T x_i + epsilon_i
        # where epsilon is a (n, ) array with values drawn from a normal distribution.
        epsilon = np.random.normal(0.0, sigma, (n, ))
        y = X.dot(w) + epsilon

        return X, y, w

def split_on_column(df, column):
    ''' Return split the given data frame df on column s.t.
    X consists of all data except column and
    y consists of only the data in column. '''
    return df.drop(column, axis=1), df[column]

def load_mnist():
    ''' Load and return the mnist training and testing datasets. '''
    data = MNIST('data/python-mnist/data/')

    # Load the data.
```

```python
    X_train, y_train = map(np.array, data.load_training())
    X_test, y_test = map(np.array, data.load_testing())

    # Reduce the data.
    X_train = X_train / 255.0
    X_test = X_test / 255.0

    return X_train, y_train, X_test, y_test

def indices(data, cols):
    indices = 0
    for c in cols:
        indices += (data == c).astype('int')
    return indices

def strip_cols(data, indices):
    ''' Return data with all columns not in cols removed. '''
    return data[indices.astype('bool')].astype('float')
```

```python
# Implementation of Gradient Descent for Problem 7.

import numpy as np
import Supplemental as sp
def gradient_descent(X, y, alpha, regularized_lambda=1E-1, w_init=None, b_init=0, delta=1E-4,
        max_iterations=1E4, verbose=False):
    # Initialize the given weights if none given.
    d = X.shape[1]
    if w_init is None:
        w_init = np.zeros(d)

    # Initialize the working weights and bias.
    w_curr, b_curr = w_init, b_init
    w_prev = w_init + float('inf')

    # Store the current iteration of the function.
    i = 0

    # Store the states of each iteration.
    w_history = []
    b_history = []
    loss_history = []

    # While not converged.
    dw = float('inf')
    while dw >= delta and i <= max_iterations:
        # Store the previous iteration for step comparison.
        # Copy to prevent side effects.
        w_prev = np.copy(w_curr)

        # Step down the gradient.
        w_curr = w_curr - alpha * sp.gradient_w(X, y, w_curr, b_curr, regularized_lambda)
        b_curr = b_curr - alpha * sp.gradient_b(X, y, w_curr, b_curr)

        # Compute the loss.
        loss = sp.gradient_loss(X, y, w_curr, b_curr, regularized_lambda)

        # Store results of current iterations.
        w_history.append(w_curr)
        b_history.append(b_curr)
        loss_history.append(loss)

        # Output results of this iteration.
        if verbose and i % 10 == 0:
            print(f'{i}\tLoss: {loss}')

        # Update for next iteration.
        dw = np.linalg.norm(w_curr - w_prev, np.inf)
        i += 1

    return w_curr, b_curr, w_history, b_history, loss_history
```

```python
# Implement Lasso for Problems 5-6.

import numpy as np

class Lasso:
    def __init__(self, regularized_lambda):
        self.rl = regularized_lambda

    def train(self, X, y, w, delta=1E-3, verbose=False):
        ''' Train the lasso using coordinate descent. '''
        n, d = X.shape
        self.w = np.zeros(d) if w is None else w

        # Let history hold the history of loss measures.
        self.history = []

        # Precompute a to speed things up considerably.
        a = 2 * np.sum(X ** 2, axis=0)

        # Let i be the current iteration.
        i = 0

        # while not converged
        w_change = float('inf')
        while w_change >= delta:
            # For measuring iteration differences.
            w_prev = np.copy(self.w)

            # b <- \frac{1}{n} \sum_{i=1}^n ( y_i - \sum_{j=1}^d w_j * x_{i,j} )
            self.b = np.mean(y - X.dot(self.w))

            # for k in {1, 2, ... d} do
            for k in range(d):
                # Access precomputed a.
                a_k = a[k]

                # Used in computing c_k
                not_k = np.arange(d) != k

                # c_k <- 2 * \sum_{i=1}^n x_{i,k} * (y_i - (b + \sum_{j \neq k} w_j * x_{i,j}) )
                c_k = 2 * np.sum(X[:, k] * (y - (self.b + X[:, not_k].dot(self.w[not_k]))), axis=0)

                # w_k is a piecewise assignment.
                # w_k <- (c_k + \lambda) / a_k   if c_k < -\lambda
                # w_k <- (c_k - \lambda) / a_k   if c_k > \lambda
                # w_k <- 0                       if c_k \in [-\lambda, \lambda]
                condlist = [c_k < -self.rl, c_k > self.rl, ]
                funclist = [(c_k + self.rl) / a_k, (c_k - self.rl) / a_k, 0]
                self.w[k] = np.float(np.piecewise(c_k, condlist, funclist))

            # Compute the loss.
            loss = self.loss(X, y)
            self.history.append(loss)

            # Output progress.
            if verbose:
                print(f'\t{i}\tLoss: {loss}')

            # Update the change in w with the new values.
            w_change = np.linalg.norm(self.w - w_prev, ord=np.inf)

            # Increment the current iteration.
            i += 1

        return self.history

    def loss(self, X, y):
```

```
        ''' Compute the lasso loss. '''
        return (np.linalg.norm(X.dot(self.w) + self.b - y)) ** 2 + self.rl * np.linalg.norm(
            self.w, ord=1)

    def predict(self, X):
        ''' Predict y_hat using the trained model. '''
        return X.dot(self.w) + self.b
```

```
# Define Performance functions for Problems 5-7.

import numpy as np

# Let the following be the threshold, below which a value
# is considered to be zero.
ZERO_THRESHOLD = 1E-14

def true_positive(actual, predicted, threshold=ZERO_THRESHOLD):
    ''' Return the count of true positives. '''
    return np.sum(np.logical_and(abs(actual)>threshold, abs(predicted)>threshold))

def true_negative(actual, predicted, threshold=ZERO_THRESHOLD):
    ''' Return the count of true negative. '''
    return np.sum(np.logical_and(abs(actual)<=threshold, abs(predicted)<=threshold))

def false_positive(actual, predicted, threshold=ZERO_THRESHOLD):
    ''' Return the count of false positives. '''
    return np.sum(np.logical_and(abs(actual)<=threshold, abs(predicted)>threshold))

def false_negative(actual, predicted, threshold=ZERO_THRESHOLD):
    ''' Return the count of false negatives. '''
    return np.sum(np.logical_and(abs(actual)>threshold, abs(predicted)<=threshold))

def fdr(actual, predicted):
  ''' Return the False Discovery Rate of the given data. '''
  tp = true_positive(actual, predicted)
  fp = false_positive(actual, predicted)
  return (fp / (tp + fp))

def tpr(actual, predicted):
  ''' Return the True Positive Rate of the given data. '''
  tp = true_positive(actual, predicted)
  fn = false_negative(actual, predicted)
  return (tp / (tp + fn))

def non_zeros(array, threshold=ZERO_THRESHOLD):
  ''' Return the count of non_zero values in the given np array. '''
  return np.sum(abs(array) > threshold)

def mse(actual, predicted):
  ''' Return the Mean Squared Error of the given data. '''
  return np.mean((actual - predicted) ** 2)

def misclass_error(actual, predicted):
  ''' Return the misclassification error. '''
  return 1 - np.mean(actual == predicted)
```

```
# Implementation of Stochastic Gradient Descent for Problem 7.

import numpy as np
import Supplemental as sp

def stochastic_gradient_descent(X, y, alpha, batch_size, regularized_lambda=1E-1, w_init=None,
        b_init=0, delta=1E-4, max_iterations=1E4, verbose=False):
 # Initialize the given weights if none given.
 n, d  = X.shape
 if w_init is None:
  w_init = np.zeros(d)

 # Initialize the working weights and bias.
```

```
    w_curr, b_curr = w_init, b_init
    w_prev = w_init + float('inf')

    # Store the current iteration of the function.
    i = 0

    # Store the states of each iteration.
    w_history = []
    b_history = []
    loss_history = []

    # While not converged.
    dw = float('inf')
    while dw >= delta and i <= max_iterations:
        # Batch the descent.
        batch_index = np.random.choice(n, batch_size)
        X_batch = X[batch_index]
        y_batch = y[batch_index]

        # Store the previous iteration for alpha comparison.
        # Copy to prevent side effects.
        w_prev = np.copy(w_curr)

        # Step down the gradient.
        w_curr = w_curr - alpha * sp.gradient_w(X_batch, y_batch, w_curr, b_curr, regularized_lambda)
        b_curr = b_curr - alpha * sp.gradient_b(X_batch, y_batch, w_curr, b_curr)

        # Compute the loss.
        loss = sp.gradient_loss(X, y, w_curr, b_curr, regularized_lambda)

        # Store results of current iterations.
        w_history.append(w_curr)
        b_history.append(b_curr)
        loss_history.append(loss)

        # Output results of this iteration.
        if verbose:
            print(f'{i}\tLoss: {loss}')

        # Update for next iteration.
        dw = np.linalg.norm(w_curr - w_prev, np.inf)
        i += 1

    return w_curr, b_curr, w_history, b_history, loss_history
```

```
# Define Supplemental functions for Problems 5-7.

import numpy as np

def max_lambda(X, y):
    ''' Return the smallest value of lambda for which the solution w_hat is entirely zero. '''
    return np.max(np.sum(2 * X * (y - np.mean(y))[:, None], axis=0))

def regularized_lambdas(max_lambda, n=20, constant=1.5):
    ''' Return a list of n regularized lambdas starting from max_lambda and
    decreasing by a constant ratio. '''
    return [(max_lambda / (constant ** i)) for i in range(n)]

def _mu(X, y, w, b):
    ''' Compute the inverse exponential. '''
    return 1 / (1 + np.exp(-y * (b + X.dot(w))))

def gradient_w(X, y, w, b, regularized_lambda=1E-1):
    ''' Return the gradient J loss function with respect to w. '''
    return np.mean(((_mu(X, y, w, b) - 1) * y)[:, None] * X, axis=0) + (2 * regularized_lambda * w)

def gradient_b(X, y, w, b):
    ''' Return the gradient J loss function with respect to b. '''
```

```
    return np.mean(((_mu(X, y, w, b) - 1) * y), axis=0)

def gradient_loss(X, y, w, b, regularized_lambda=1E-1):
    ''' Return the loss function J. '''
    return np.mean(np.log(1 + np.exp(-y * (b + X.dot(w))))) + (regularized_lambda * w.dot(w))

def gradient_predict(X, w, b):
    ''' Perform a prediction using the gradient weights. '''
    return np.sign(X.dot(w) + b)
```