Homework 3 Solutions

Due April 24, 2020 by 11:59pm

Instructions: Upload your answers to the questions below to Canvas. Submit the answers to the questions in a PDF file and your code in a (single) separate file, including for the data competition exercise. Be sure to comment your code to indicate which lines of your code correspond to which question part. There are 3 study assignments and 2 exercises in this homework.

Reading Assignments

- Review Lecture 3.
- Review Computer Lab. 3 in *canvas.uw.edu/courses/1371621/pages/course-materials* .
- Read and explore *distill.pub/2017/momentum/* .

1 Exercise 1

In this exercise, you will implement in **Python** a first version of *your own fast gradient algorithm* to solve the ℓ_2^2-regularized logistic regression problem.

Recall from the lectures that the logistic regression problem writes as

$$\min_{\beta \in \mathbb{R}^d} F(\beta) := \frac{1}{n} \sum_{i=1}^{n} \log\left(1 + \exp(-y_i\, x_i^T \beta)\right) \; + \lambda \|\beta\|_2^2 \; . \tag{1}$$

We use here the machine learning convention for the labels that is $y_i \in \{-1, +1\}$.

1.1 Fast Gradient

The fast gradient algorithm is outlined in Algorithm 1. The algorithm requires a subroutine that computes the gradient for any β.

- Assume that $d = 1$ and $n = 1$. The sample is then of size 1 and boils down to just (x, y). The function F writes simply as

$$F(\beta) = \log(1 + \exp(-yx\,\beta) \; + \lambda\beta^2 \; . \tag{2}$$

Compute and write down the gradient ∇F of F.

$$\nabla F(\beta) = -yx\frac{\exp(-y\,x\beta)}{1+\exp(-y\,x\beta)} + 2\lambda\beta$$

- Assume now that $d > 1$ and $n > 1$. Using the previous result and the linearity of differentiation, compute and write down the gradient $\nabla F(\beta)$ of F.

$$\nabla F(\beta) = \frac{1}{n}\sum_{i=1}^{n} -y_i x_i \frac{\exp(-y_i\,x_i^T\beta)}{1+\exp(-y_i\,x_i^T\beta)} + 2\lambda\beta$$

- Consider the Spam dataset from *The Elements of Statistical Learning* (You can get it here: https://web.stanford.edu/~hastie/ElemStatLearn/). Standardize the data (i.e., center the features and divide them by their standard deviation, and also change the output labels to +/- 1).

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.linalg
import sklearn.linear_model
import sklearn.preprocessing

# Part (c): Read in the data, standardize it
spam = pd.read_table('https://web.stanford.edu/~hastie/'
                     'ElemStatLearn/datasets/spam.data',
                     sep=' ', header=None)
test_indicator = pd.read_table('https://web.stanford.edu/'
                               '~hastie/ElemStatLearn/datasets/'
                               'spam.traintest', sep=' ',
                               header=None)

x = np.asarray(spam)[:, 0:-1]
y = np.asarray(spam)[:, -1]*2 - 1  # Convert to +/- 1
test_indicator = np.array(test_indicator).T[0]

# Divide the data into train, test sets
x_train = x[test_indicator == 0, :]
x_test = x[test_indicator == 1, :]
y_train = y[test_indicator == 0]
y_test = y[test_indicator == 1]

# Standardize the data.
scaler = sklearn.preprocessing.StandardScaler()
scaler.fit(x_train)
```

```python
x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)

# Keep track of the number of samples and dimension of each sample
n_train = len(y_train)
n_test = len(y_test)
d = np.size(x, 1)
```

- Write a function *computegrad* that computes and returns $\nabla F(\beta)$ for any β.

```python
def computegrad(beta, lambduh, x=x_train, y=y_train):
    yx = y[:, np.newaxis]*x
    denom = 1+np.exp(-yx.dot(beta))
    grad = 1/len(y)*np.sum(-yx*np.exp(-yx.dot(beta[:, np.newaxis]))/
                           denom[:, np.newaxis], axis=0) \
           + 2*lambduh*beta
    return grad
```

- Write a function *backtracking* that implements the backtracking rule.

```python
def objective(beta, lambduh, x=x_train, y=y_train):
    return 1/len(y) * np.sum(np.log(1 + np.exp(-y*x.dot(beta)))) \
        + lambduh * np.linalg.norm(beta)**2
```

```python
def backtracking(beta, lambduh, eta=1, alpha=0.5, betaparam=0.8,
                 maxiter=100, x=x_train, y=y_train):
    grad_beta = computegrad(beta, lambduh, x=x, y=y)
    norm_grad_beta = np.linalg.norm(grad_beta)
    found_eta = 0
    num_iters = 0
    while found_eta == 0 and num_iters < maxiter:
        if objective(beta - eta * grad_beta, lambduh, x=x, y=y) < \
                        objective(beta, lambduh, x=x, y=y)\
                        - alpha * eta * norm_grad_beta ** 2:
            found_eta = 1
        elif num_iters == maxiter:
            raise ('Max number of iterations of backtracking'
                   ' line search reached')
        else:
            eta *= betaparam
            num_iters += 1
    return eta
```

- Write a function *graddescent* that implements the gradient descent algorithm with the backtracking rule to tune the step-size. The function *graddescent* calls *computegrad* and *backtracking* as subroutines. The function takes as input the initial point, the initial step-size value, and the target accuracy ε. The stopping criterion is $\|\nabla F\| \leq \varepsilon$.

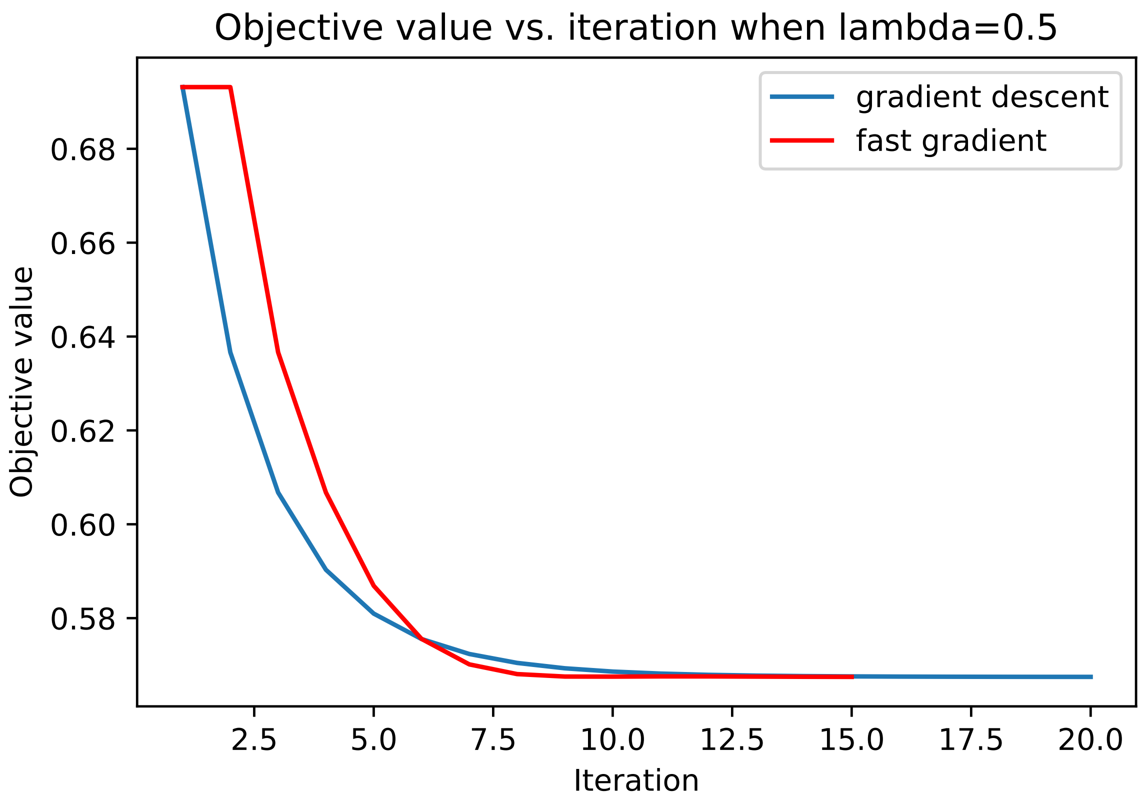
```python
def graddescent(beta_init, lambduh, eta_init,
                x=x_train, y=y_train, eps=5.1**-3):
    beta = beta_init
    grad_beta = computegrad(beta, lambduh, x=x, y=y)
    beta_vals = beta
    num_iters = 0
    while np.linalg.norm(grad_beta) > eps:
        eta = backtracking(beta, lambduh, eta=eta_init, x=x, y=y)
        beta = beta - eta*grad_beta
        # Store all of the places we step to
        beta_vals = np.vstack((beta_vals, beta))
        grad_beta = computegrad(beta, lambduh, x=x, y=y)
        num_iters += 1
    return beta_vals
```

- Write a function *fastgradalgo* that implements the fast gradient algorithm described in Algorithm 1. The function *fastgradalgo* calls *computegrad* and *backtracking* as subroutines. The function takes as input the initial step-size value for the backtracking rule and the target accuracy ε. The stopping criterion is $\|\nabla F\| \leq \varepsilon$.

```python
def fastgradalgo(beta_init, theta_init, lambduh, eta,
                 x=x_train, y=y_train, eps=5.1**-3):
    beta = beta_init
    theta = theta_init
    grad_theta = computegrad(theta, lambduh, x=x, y=y)
    grad_beta = computegrad(beta, lambduh, x=x, y=y)
    beta_vals = beta
    theta_vals = theta
    num_iters = 0
    while np.linalg.norm(grad_beta) > eps:
        eta = backtracking(theta, lambduh, eta=eta, x=x, y=y)
        beta_new = theta - eta*grad_theta
        theta = beta_new + num_iters/(num_iters+3)*(beta_new-beta)
        # Store all of the places we step to
        beta_vals = np.vstack((beta_vals, beta))
        theta_vals = np.vstack((theta_vals, theta))
        grad_theta = computegrad(theta, lambduh, x=x, y=y)
        grad_beta = computegrad(beta, lambduh, x=x, y=y)
        beta = beta_new
        num_iters += 1
    return beta_vals
```

- Use the estimate described in the course to initialize the step-size. Set the target accuracy to $\varepsilon = 5.10^{-3}$. Run *graddescent* and *fastgradalgo* on the training set of the Spam dataset for $\lambda = 0.5$. Plot the curve of the objective values $F(\beta_t)$ for both algorithms versus the iteration counter t (use different colors). What do you observe?

```python
def objective_plot(betas_gd, betas_fg, lambduh, x=x_train,
                   y=y_train, save_file=''):
    num_points_gd = np.size(betas_gd, 0)
    objs_gd = np.zeros(num_points_gd)
    num_points_fg = np.size(betas_fg, 0)
    objs_fg = np.zeros(num_points_fg)
    objective(betas_fg[0, :], lambduh, x=x, y=y)
    for i in range(num_points_gd):
        objs_gd[i] = objective(betas_gd[i, :], lambduh, x=x, y=y)
    for i in range(num_points_fg):
        objs_fg[i] = objective(betas_fg[i, :], lambduh, x=x, y=y)
    fig, ax = plt.subplots()
    ax.plot(range(1, num_points_gd + 1), objs_gd,
           label='gradient descent')
    ax.plot(range(1, num_points_fg + 1), objs_fg, c='red',
            label='fast gradient')
    plt.xlabel('Iteration')
    plt.ylabel('Objective value')
    plt.title('Objective value vs. iteration when lambda='+str(lambduh))
    ax.legend(loc='upper right')
    if not save_file:
        plt.show()
    else:
        plt.savefig(save_file)


lambduh = 0.5
beta_init = np.zeros(d)
theta_init = np.zeros(d)
# See slide 26 in the lecture 3 slides for how to
# initialize the step size
eta_init = 1/(scipy.linalg.eigh(1/len(y_train)*x_train.T.dot(x_train),
                                eigvals=(d-1, d-1),
                                eigvals_only=True)[0]+lambduh)
maxiter = 1000
betas_grad = graddescent(beta_init, lambduh, eta_init)
betas_fastgrad = fastgradalgo(beta_init, theta_init, lambduh,
                              eta_init)
objective_plot(betas_grad, betas_fastgrad, lambduh)
```



Fast gradient converges faster, although has a slightly worse objective value at the first six or so iterations.

- Denote by β_T the final iterate of your fast gradient algorithm. Compare β_T to the $\beta^\star$ found by *scikit-learn*. Compare the objective value for β_T to the one for $\beta^\star$. What do you observe?

```
lr = sklearn.linear_model.LogisticRegression(penalty='l2',
                                             C=1/(2*lambduh*n_train),
                                             fit_intercept=False,
                                             tol=10e-8, max_iter=1000)
lr.fit(x_train, y_train)
print(lr.coef_)
print(betas_fastgrad[-1, :])

print(objective(betas_fastgrad[-1, :], lambduh))
print(objective(lr.coef_.flatten(), lambduh))
```

```
[[ 0.02108078 -0.01487315  0.0543084   0.01991117  0.06748598
0.0653377
   0.10631408  0.06047205  0.05532727  0.0327294   0.05976812
-0.00816259
   0.02702136  0.0152748   0.04929311  0.10530449  0.07345384
0.05789425
```

```
   0.07061852  0.05335277  0.10958426  0.03712076  0.09590903
0.06746536
  -0.06450613 -0.05315548 -0.05201082 -0.03511908 -0.02640437
-0.03441331
  -0.02012504 -0.01433024 -0.03249274 -0.01442876 -0.02478369
-0.02031716
  -0.04354537 -0.01165974 -0.03610122  0.00101422 -0.02483072
-0.03793039
  -0.03254773 -0.02808211 -0.04384952 -0.0459123  -0.013809
-0.02565912
  -0.01894061 -0.02027317 -0.01575     0.06312524  0.09558183
0.02136077
   0.02873048  0.04994261  0.06379162]]
[ 0.02036246 -0.01519786  0.05412132  0.02037302  0.06820558
0.06556996
  0.10810535  0.06117922  0.05514696  0.03258744  0.05940849
-0.0090832
  0.02656068  0.01516817  0.04920245  0.10706747  0.07388488
0.05822514
  0.07047091  0.05354828  0.11008365  0.03815258  0.09691153
0.06819826
 -0.06474485 -0.05295755 -0.05237224 -0.03476519 -0.02610855
-0.03418103
 -0.0195547  -0.01368149 -0.03275634 -0.01378832 -0.02445393
-0.01952507
 -0.04328419 -0.01195611 -0.03624059  0.001739   -0.02473479
-0.03826558
 -0.0323988  -0.02842153 -0.04454288 -0.04643654 -0.01397686
-0.02593263
 -0.01938131 -0.02039573 -0.01561916  0.0638754   0.0968899
0.02172778
  0.02882172  0.05017138  0.06416374]
0.5674825556310084
0.5674697232274132
```

They're pretty close.

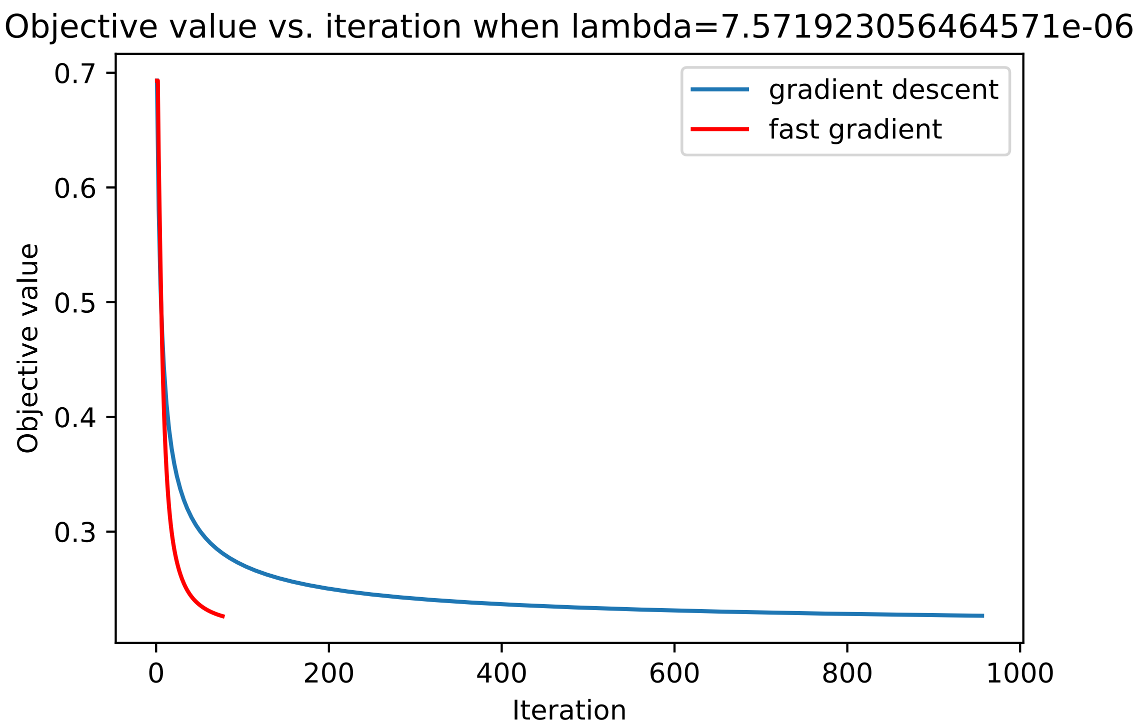
- Run cross-validation on the training set of the Spam dataset using *scikit-learn* to find the optimal value of λ. Run *graddescent* and *fastgradalgo* to optimize the objective with that value of λ. Plot the curve of the objective values $F(\beta_t)$ for both algorithms versus the iteration counter t. Plot the misclassification error on the training set for both algorithms versus the iteration counter t. Plot the misclassification error on the test set for both algorithms versus the iteration counter t. What do you observe?

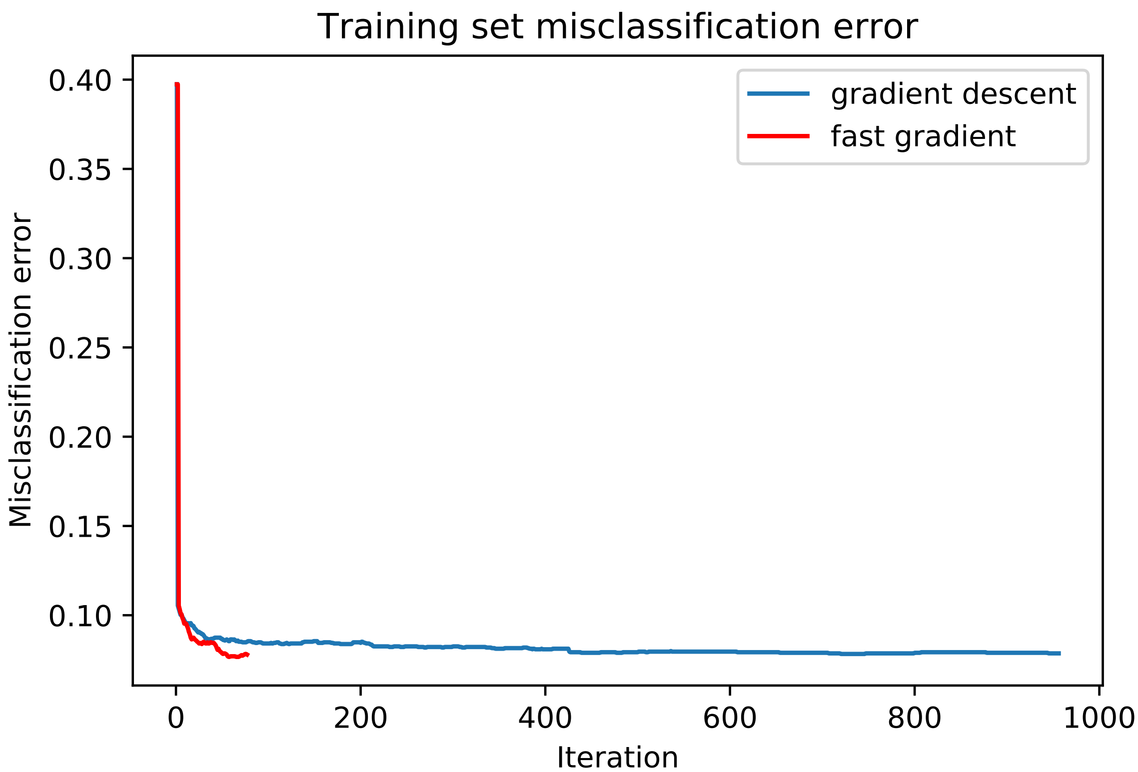
```
def compute_misclassification_error(beta_opt, x, y):
    y_pred = 1/(1+np.exp(-x.dot(beta_opt))) > 0.5
    y_pred = y_pred*2 - 1  # Convert to +/- 1
```

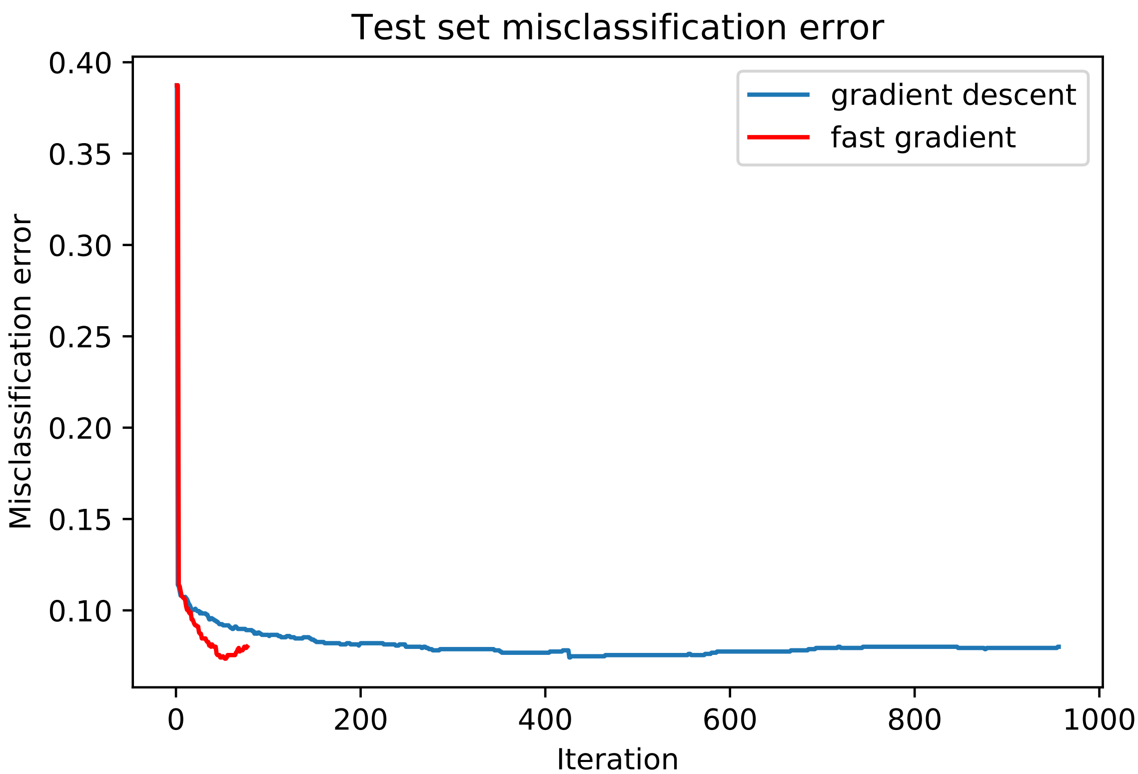
```
    return np.mean(y_pred != y)


def plot_misclassification_error(betas_grad, betas_fastgrad,
                                 x, y, save_file='', title=''):
    niter_grad = np.size(betas_grad, 0)
    error_grad = np.zeros(niter_grad)
    niter_fg = np.size(betas_fastgrad, 0)
    error_fastgrad = np.zeros(niter_fg)
    for i in range(niter_grad):
        error_grad[i] = compute_misclassification_error(
            betas_grad[i, :], x, y)
    for i in range(niter_fg):
        error_fastgrad[i] = compute_misclassification_error(
            betas_fastgrad[i, :], x, y)
    fig, ax = plt.subplots()
    ax.plot(range(1, niter_grad + 1), error_grad,
            label='gradient descent')
    ax.plot(range(1, niter_fg + 1), error_fastgrad, c='red',
            label='fast gradient')
    plt.xlabel('Iteration')
    plt.ylabel('Misclassification error')
    if title:
        plt.title(title)
    ax.legend(loc='upper right')
    if not save_file:
        plt.show()
    else:
        plt.savefig(save_file)

lr_cv = sklearn.linear_model.LogisticRegressionCV(penalty='l2',
                                                  fit_intercept=False,
                                                  tol=10e-8,
                                                  max_iter=1000)
lr_cv.fit(x_train, y_train)
optimal_lambda = 1/(2*lr_cv.C_[0]*len(x_train))
print('Optimal C=', lr_cv.C_[0])
print('Optimal lambda=', optimal_lambda)

eta_init = 1/(scipy.linalg.eigh(1/len(y_train)*x_train.T.dot(x_train),
                                eigvals=(d-1, d-1),
                                eigvals_only=True)[0]+optimal_lambda)

betas_grad = graddescent(beta_init, optimal_lambda, eta_init)
betas_fastgrad = fastgradalgo(beta_init, theta_init, optimal_lambda,
                              eta_init)
```

```
objective_plot(betas_grad, betas_fastgrad, optimal_lambda,
               save_file='hw3_q1_part_j_output1.png')
plot_misclassification_error(betas_grad, betas_fastgrad, x_train,
                             y_train,
                             save_file=None,
                             title='Training set misclassification '
                                   'error')
plot_misclassification_error(betas_grad, betas_fastgrad, x_test,
                             y_test,
                             save_file=None,
                             title='Test set misclassification error')
```

```
Optimal C= 21.54434690031882
Optimal lambda= 7.571923056464571e-06
```







Fast gradient converges faster.

Algorithm 1 Fast Gradient Algorithm

input step-size η_0, target accuracy ε
initialization $\beta_0 = 0$, $\theta_0 = 0$
repeat for $t = 0, 1, 2, \ldots$
Find η_t with backtracking rule
$\beta_{t+1} = \theta_t - \eta_t \nabla F(\theta_t)$
$\theta_{t+1} = \beta_{t+1} + \frac{t}{t+3}(\beta_{t+1} - \beta_t)$
until the stopping criterion $\|\nabla F\| \leq \varepsilon$.

2 Exercise 2

Suppose we estimate the regression coefficients in a logistic regression model by minimizing

$$F(\beta) := \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp(-y_i\, x_i^T \beta)\right) + \lambda\|\beta\|_2^2$$

for a particular value of λ. For parts (a) through (e), indicate which of (i) through (v) is correct. Justify your answer.

(a) As we increase λ from 0, the misclassification error on the training set will:

(i) Increase initially, and then eventually start decreasing in an inverted U shape.
(ii) Decrease initially, and then eventually start increasing in a U shape.
(iii) Steadily increase.
(iv) Steadily decrease.
(v) Remain constant.
(vi) Zigzag in mysterious ways.

(iii) It will steadily increase. It is the first term in the regularized logistic regression objective function that minimizes the misclassification error. As λ increases, the first term necessarily becomes larger at the optimal β, thereby increasing the misclassification error.

(b) Repeat (a) for the misclassification error on a large dataset of unseen data draw from the same probability distribution as the training set.

(ii) It will decrease initially, and then eventually start increasing in a U shape. The misclassification error on this dataset depends on the variance and the bias. When $\lambda = 0$, the variance will likely be large if d is large because the model will overfit the training data. As λ increases, the variance will decrease without the bias decreasing too much. Similarly to Figure 6.5 in the ISL textbook, there will be some value of λ for which the classification error on this dataset is smallest, and anything larger or smaller than that λ will lead to a higher classification error.