Homework 3

Due April 26, 2019 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 is 1 reading assignment and 5 exercises in this homework. Exercise 4 is the first milestone of the data competition. Exercise 5 is optional, not to be turned in.

Reading Assignment

Read Sec. 4.4 to 4.5 in *The Elements of Statistical Learning*.

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)
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.

```
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.

```
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
```

```
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
    iter = 0
    while found_eta == 0 and iter < 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 iter == maxiter:
            raise ('Max number of iterations of backtracking'
                   ' line search reached')
        else:
            eta *= betaparam
            iter += 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 $\|\nabla F\| \leq \varepsilon$.

```
def graddescent(beta_init, lambduh, eta_init,
                x=x_train, y=y_train, eps=1e-4):
```

```
    beta = beta_init
    grad_beta = computegrad(beta, lambduh, x=x, y=y)
    beta_vals = beta
    iter = 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)
        iter += 1
    return beta_vals
```

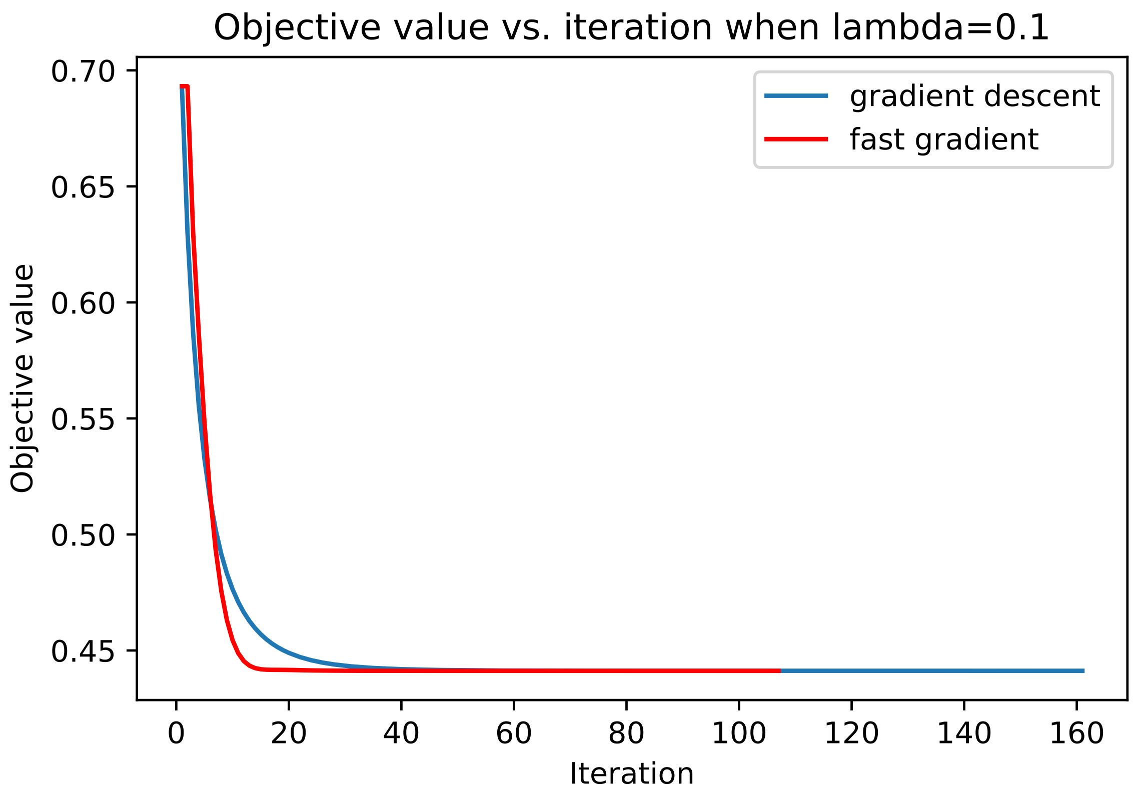
- Write a function *fastgradalgo* that implements the fast gradient algorithm described in Algorithm 1. The function *fastgradalgo* calls *computegrad* and *backtracking* as subroutines. The function takes as input the initial step-size value for the backtracking rule and the target accuracy ε. The stopping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

```
def fastgradalgo(beta_init, theta_init, lambduh, eta_init,
                 x=x_train, y=y_train, eps=1e-4):
    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
    iter = 0
    while np.linalg.norm(grad_beta) > eps:
        eta = backtracking(theta, lambduh, eta=eta_init, x=x, y=y)
        beta_new = theta - eta*grad_theta
        theta = beta_new + iter/(iter+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
        iter += 1
    return beta_vals
```

- Use the estimate described in the course to initialize the step-size. Set the target accuracy to $\varepsilon = 10^{-4}$. Run *graddescent* and *fastgradalgo* on the training set of the Spam dataset for $\lambda = 0.1$. 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?

```
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 = 0.1
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 ten 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?
  As in the last homework, we can find the relationship between C in the scikit-learn objective function and λ. This time we have $C = 1/(2\lambda n)$.

```
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))
```

```
[[ 0.02117345 -0.03877587  0.09773771  0.05503819  0.15256401
0.13534366
   0.28352681  0.15114461  0.11280873  0.06183421  0.10502022
-0.04219962
```

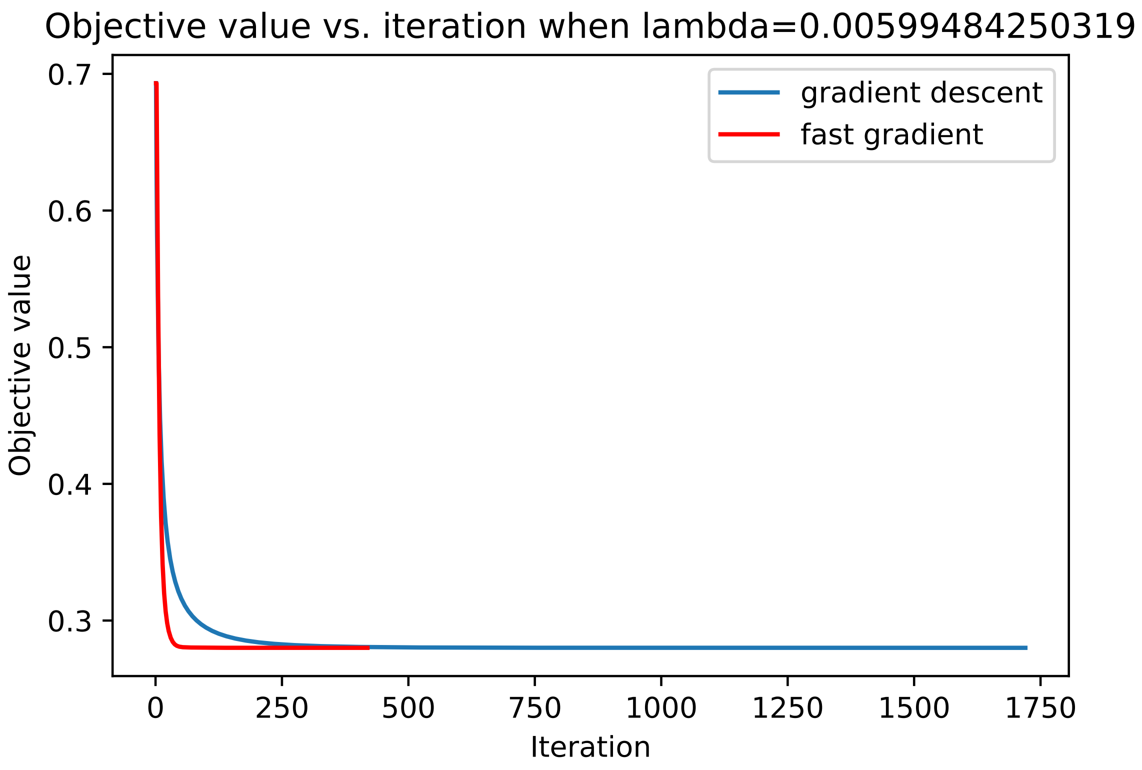
```
   0.03472887  0.02859174  0.10917878  0.27363777  0.17373109
0.12517347
   0.1290986   0.12243222  0.2240522   0.10833277  0.24549208
0.16655949
  -0.14754037 -0.10493049 -0.1149898  -0.06087818 -0.04741795
-0.06626914
  -0.02882511 -0.0151532  -0.07621369 -0.0159425  -0.04245255
-0.01357162
  -0.07700499 -0.03415877 -0.07954035  0.01733619 -0.04772964
-0.09082759
  -0.06004341 -0.06969745 -0.11271606 -0.11263913 -0.0321344
-0.06206727
  -0.05727232 -0.0425365  -0.02823825  0.15484338  0.26205747
0.05893876
   0.06871901  0.12509356  0.14818851]]
[ 0.02112794 -0.03881638  0.09771584  0.0550708   0.15258703
0.13536117
  0.28345094  0.15120593  0.11279985  0.06179691  0.10514946
-0.04223437
  0.03468974  0.0285735   0.10916229  0.27366578  0.17379164
0.12522376
  0.12908499  0.12249728  0.22409768  0.10827572  0.24538103
0.16645622
 -0.14750102 -0.10495349 -0.11497942 -0.06092515 -0.04744043
-0.06626905
 -0.02885572 -0.01516067 -0.07623068 -0.01595713 -0.04248181
-0.01363739
 -0.07704792 -0.03418873 -0.07950899  0.01733366 -0.04776409
-0.09082228
 -0.06002897 -0.06969176 -0.11266884 -0.11261225 -0.03214546
-0.06205871
 -0.05729261 -0.04250495 -0.02824788  0.15483327  0.26198898
0.05894004
  0.0686914   0.12513369  0.14817976]
0.441222693823
0.441222678066
```

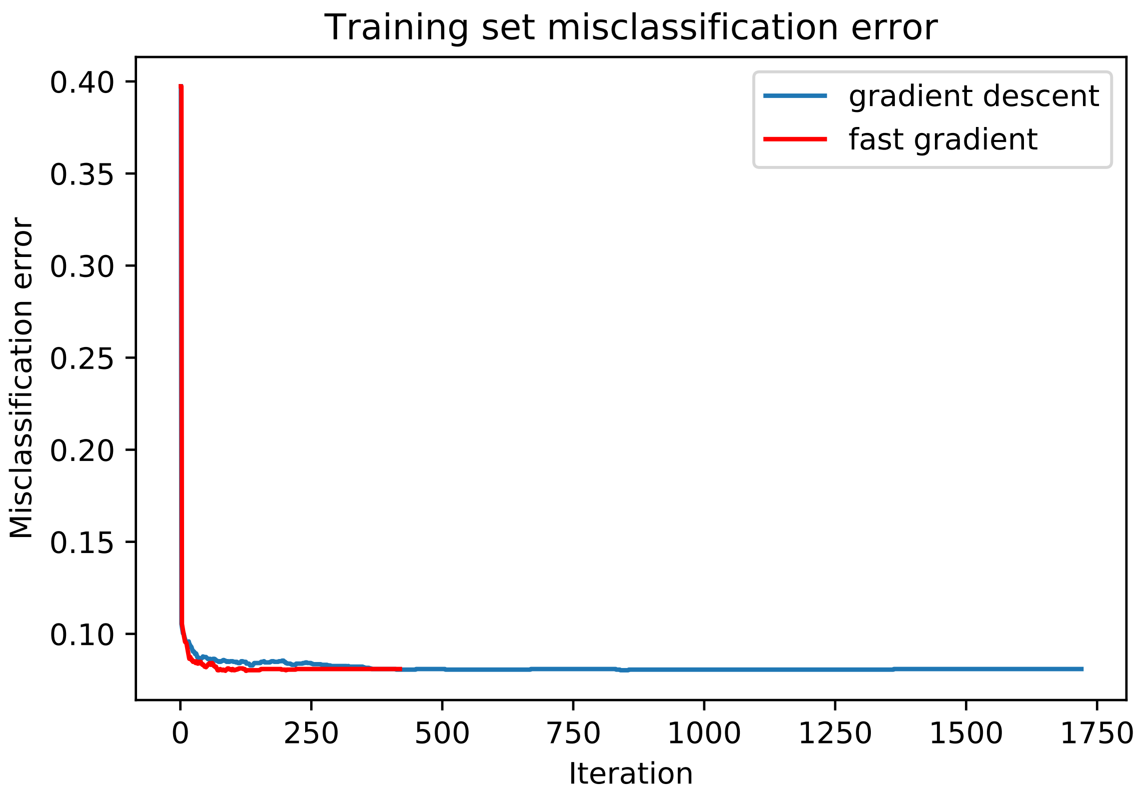
They're very close.

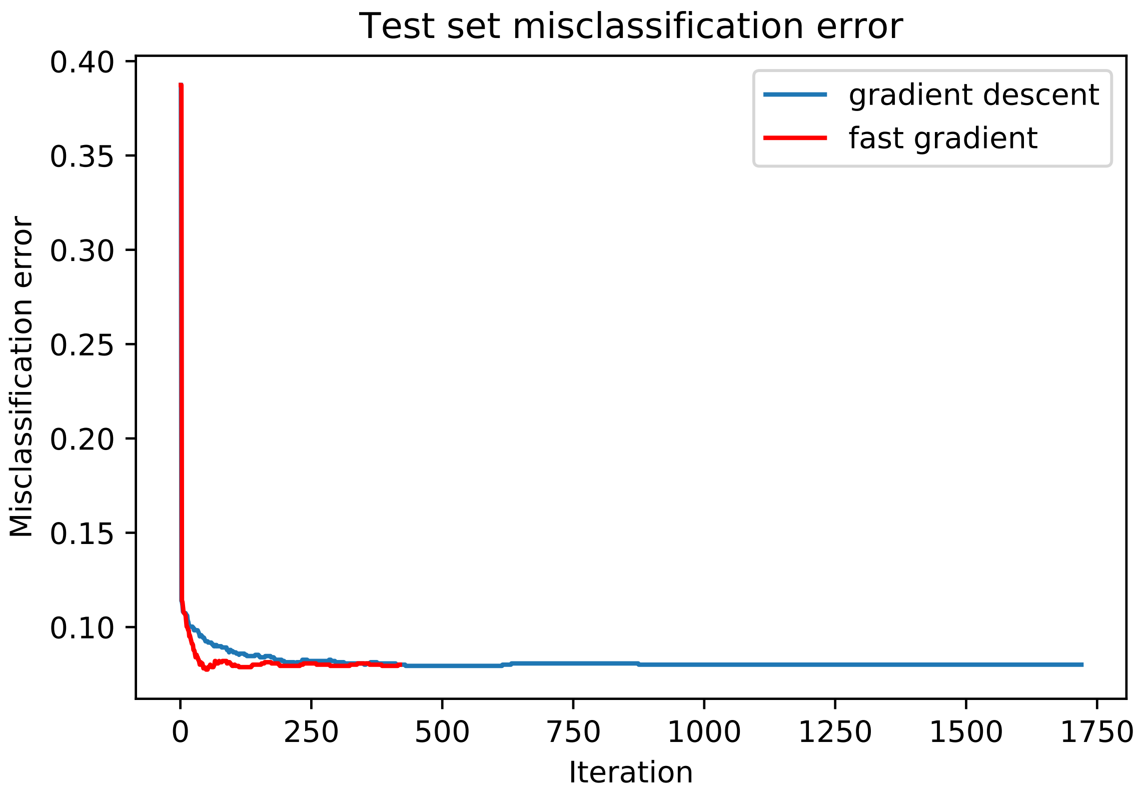
- Run cross-validation on the training set of the Spam dataset using *scikit-learn* to find the optimal value of λ. Run *graddescent* and *fastgradalgo* to optimize the objective with that value of λ. Plot the curve of the objective values $F(\beta_t)$ for both algorithms versus the iteration counter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

```python
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 = lr_cv.C_[0]
print('Optimal lambda=', 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 lambda= 0.00599484250319
```




Training set misclassification error
Misclassification error
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0
250
500
750
1000
1250
1500
1750
Iteration
gradient descent
fast gradient


Test set misclassification error
Misclassification error
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0
250
500
750
1000
1250
1500
1750
Iteration
gradient descent
fast gradient

Algorithm 1 Fast Gradient Algorithm

```
input   step-size $\eta_0$, target accuracy $\varepsilon$
initialization   $\beta_0 = 0$, $\theta_0 = 0$
repeat for $t = 0, 1, 2, \ldots$
Find $\eta_t$ 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 (b), indicate which of (i) through (v) is correct. Justify your answer.

(a) As we increase λ from 0, the misclassification error on the test 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.

   (ii) It will decrease initially, and then eventually start increasing in a U shape. The test misclassification error 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 test classification error is smallest, and anything larger or smaller than that λ will lead to a higher test classification error.

(b) Repeat (a) for the misclassification error on the training set.
   (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.

3 Exercise 3

In this exercise, you will use Amazon Web Services (AWS) to run a nearest neighbors algorithm on data from the Sloan Digital Sky Survey. The goals of this exercise are:

1. To teach you how to use AWS's Elastic Compute Cloud (EC2) and Simple Storage Service (S3)
2. To show you that computing with GPUs can be much faster than computing with CPUs
3. To demonstrate a fast algorithm for computing nearest neighbors (from [**?**])

Background: The Sloan Digital Sky Survey (SDSS) has gathered data on many objects in the sky. The data set you will use in this assignment consists of (1) a "training" set of photometric data on known astronomical objects; and (2) a "test" set of additional photometric data on more objects. Astronomers often need to determine which objects in a new data set are worth further examination because their telescope time is very limited. By finding the nearest neighbors in the training set to each object in the test set, they can tell whether the object is interesting or not. For example, if an object in the test set is closest to sun-like stars, then they might find that one very boring and not follow up on it.

Instructions: Following the AWS tutorial for help, do the following:

1. Create a p2.xlarge spot instance with the "Deep Learning AMI (Ubuntu) Version 22.0" AMI. If this instance type doesn't work, try another one of the gpu instances. You need to use a GPU instance for this assignment.
2. Install Swig and bufferkdtree on your instance.
3. On your instance, run the astronomy example from here `https://github.com/gieseke/bufferkdtree/tree/master/examples` and save all of the output to the file "output.txt". **Hint:** You will need to transfer both astronomy.py and generate.py on EC2. In addition, in the file astronomy.py you will need to change the line

   ```
   plat_dev_ids = {0:[0,1,2,3]}
   ```

   to

   ```
   plat_dev_ids = {0:[0]}
   ```

   The data will automatically download the first time you run it **after user input.** Therefore, do not try to save the output to a file via > the first time you run it or else it will hang indefinitely. Either use the `tee` command (`https://en.wikipedia.org/wiki/Tee_(command)`), run the script once before using > or copy and paste the output to a file. If you are still having issues with it hanging when saving directly to a file, check the output file. It's possible that it finished writing out the output.
4. Transfer the file output.txt to an S3 bucket.
5. Go to the S3 interface `https://console.aws.amazon.com/s3/home?region=us-east-1` and make that file public.
6. In your homework submission please include the following:

(a) The fitting and testing times on both the CPU and GPU versions. You can find these in the output.

(b) The url from the previous step (Check to make sure you successfully made it public!)

(c) A statement of any problems you encountered during this exercise and how you overcame them (or if you didn't).

(d) How long it took you to complete this exercise (for our reference–we're not grading you on how long it took).

Remember to terminate your instance when you are done with it!
Selected issues some of you encountered and your solutions were:

- Can't install swig. Solution: Run apt-get update.
- Problems running it with Python 3. Solution: Run with Python 2.
- Max spot instance count exceeded. Solution: Email AWS support and request an increase.
- Code got stuck when writing output to file. Solution: (1) Print the output to the console and copy/paste it; (2) Run the code twice, the first time not saving the output to a file and the next time saving the output to a file; or (3) Use the "tee" command to both print the output to the console and save it to a file.
- "Launch Failed - Your account is currently being verified. Verification normally takes less than 2 hours." Solution: Email AWS support.
- Unable to paste the content from astronomy.py directly into a new file in the command line in Ubuntu. Solution: Download the file from Github and then moved it onto the virtual machine via scp
- Couldn't connect to a p2.xlarge spot instance. Solution: Used g2.xlarge or g2.2xlarge instead.
- Losing internet connection briefly when connected to the instance. Solution: Reconnect to the instance.
- The Python script never appeared to complete. (Partial) solution: The output had already been saved in the file anyway.

4 Exercise 4

Read the announcement "Data Competition, Part 1" released on Canvas. We strongly recommend you perform this task on AWS. You will use *your own ℓ_2^2-regularized logistic regression* for this exercise. After completing this exercise, submit your predictions to the data competition Kaggle website.

- Download the data for the Kaggle competition. Run the script `extract_features.py` to extract features from the images. This script was written in Python 3 and depends on the library PyTorch.
- Pick two classes of your choice from the dataset. Train an ℓ_2^2-regularized logistic regression classifier on the training set using your own fast gradient algorithm with $\lambda = 1$. Be sure to use the features you generated above rather than the raw image features. Plot, with different colors, the *misclassification error* on the training set and on the validation set vs iterations.
- Find the value of the regularization parameter λ using cross-validation; you may use scikit-learn's built-in functions for this purpose. Train an ℓ_2^2-regularized logistic regression classifier on the training set using your own fast gradient algorithm with that value of λ found by cross-validation. Plot, with different colors, the *misclassification error* on the training set and on the validation set vs. iterations.

5 Optional Exercise

It is well-known that ridge regression tends to give similar coefficient values to correlated variables, whereas the lasso may give quite different coefficient values to correlated variables. We will now explore this property in a very simple setting.

Suppose that $n = 2, p = 2, x_{11} = x_{12}, x_{21} = x_{22}$. Furthermore, suppose that $y_1 + y_2 = 0$ and $x_{11} + x_{21} = 0$ and $x_{12} + x_{22} = 0$, so that the estimate for the intercept in a least squares, ridge regression, or lasso model is zero: $\hat{\beta}_0 = 0$.

(a) Write out the ridge regression optimization problem in this setting.
The ridge regression problem is given by

$$\min_{\beta_1,\beta_2} \frac{1}{2}\sum_{i=1}^{2}\left(y_i - \sum_{j=1}^{2}\beta_j x_{ij}\right) + \lambda\sum_{j=1}^{2}\beta_j^2$$

$$\iff \min_{\beta_1,\beta_2} \frac{1}{2}\left[(y_1 - \beta_1 x_{11} - \beta_2 x_{12})^2 + (y_1 - \beta_1 x_{21} - \beta_2 x_{22})^2\right] + \lambda(\beta_1^2 + \beta_2^2)$$

$$\iff \min_{\beta_1,\beta_2} \frac{1}{2}\left[(y_1 - (\beta_1 + \beta_2)x_{11})^2 + (y_1 - (\beta_1 + \beta_2)x_{22})^2\right] + \lambda(\beta_1^2 + \beta_2^2).$$

(b) Argue that in this setting, the ridge coefficient estimates satisfy $\beta_1 = \beta_2$.
The first order conditions for a minimum are found by taking the partial derivatives with respect to β_1 and β_2 and setting them equal to zero:

$$-x_{11}(y_1 - (\beta_1 + \beta_2)x_{11}) - x_{22}(y_1 - (\beta_1 + \beta_2)x_{22}) + \lambda\beta_1 = 0$$
$$-x_{11}(y_1 - (\beta_1 + \beta_2)x_{11}) - x_{22}(y_1 - (\beta_1 + \beta_2)x_{22}) + \lambda\beta_2 = 0.$$

Subtracting the two equations above yields

$$\lambda(\beta_1 - \beta_2) = 0.$$

Hence, assuming $\lambda > 0$, we must have $\beta_1 = \beta_2$.