Homework 2 Solutions

Due Friday April 17, 2020, by 11:59pm

Instructions: All coding exercises must be completed in Python. 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. Be sure to comment your code to indicate which lines of your code correspond to which question part. There are 3 study assignments and 3 exercises in this homework.

Reading Assignments

- Study Computer Lab. 2 in *canvas.uw.edu/courses/1371621/pages/course-materials* .
- Study Sec. 4.1 to 4.4.2 and Sec. 7.10 in *The Elements of Statistical Learning*.
- Study Sec. 5.1 to 5.5 in *Mathematics of Machine Learning*.

1 Exercise 1

In this exercise, you will implement a first version of *your own gradient descent algorithm* to solve the ridge regression problem. Throughout the homeworks, you will keep improving and extending your gradient descent optimization algorithm. In this homework, you will implement a basic version of the algorithm.

Recall from Week 1 and Week 2 Lectures that the ridge regression problem writes as

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

that is, if you expand,

$$\min_{\beta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{d} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{d} \beta_j^2 \; . \tag{2}$$

1.1 Remarks

Several remarks are in order.

Normalization Note that there is a $1/n$ normalization factor in the empirical risk term in the equations. Note also that there is a λ multiplicative factor in the regularization penalty term in the equations. Sometimes, in articles, you may see the normalization $\lambda/2$ instead for the ℓ_2^2-regularization penalty. This is convenient when you compute the gradient of that term because the 2 and the $1/2$ cancel.

You can actually normalize the terms any way you want *as long as you are consistent all the way through* in your mathematical derivations, your codes, and your experiments, especially when you search over parameters.

So here is my general advice:

- do normalize the empirical risk term so that it is an average, not a sum; this normalization will be important for large scale problems where the sum can become very large. Indeed, with the normalization, the average remains of the same order of magnitude regardless of the number of terms in the sum.
- check what optimization problem exactly is solved when you use a library, so you can compare your solution to the optimization problem to the solution found by the library and compare the optimal value of the regularization found by your grid search to the one found the library's grid search.

Intercept It is common in traditional statistics and machine learning books and libraries to include an intercept β_0 in the statistical model. Having a separate intercept coefficient is actually not that important, and provably so, especially if the data was properly centered and standardized beforehand.

There is actually a simple way to bypass the issue of having a separate intercept coefficient by adding a constant variable 1 in the variables. See Sec. 2.3.1 of *The Elements of Statistical Learning*. So the d variables in the equations correspond to the $(d-1)$ original variables plus 1 dummy variable equal to 1.

1.2 Gradient descent

The gradient descent algorithm is an iterative algorithm that is able to solve differentiable optimization problems such as (1). Define

$$F(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i^T\beta)^2 \ + \lambda\|\beta\|_2^2 \ . \tag{3}$$

Gradient descent generates a sequence of iterates[1] (β_t) that converges to the optimal solution $\beta^\star$ of (1). The optimal solution of (1) is defined as

$$F(\beta^\star) = \min_{\beta\in\mathbb{R}^d} F(\beta) \ . \tag{4}$$

Gradient descent is outlined in Algorithm 1. The algorithm requires a sub-routine that computes the gradient for any β. The algorithm also takes as input the value of the constant step-size η.

[1]The subscript t refers to the iteration counter here, not to the coordinates of the vector 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

$$F(\beta) = (y - x\,\beta)^2 \; + \lambda\beta^2 \,. \tag{5}$$

  Compute and write down the gradient ∇F of F.
  $\nabla F(\beta) = -2x(y - x\beta) + 2\lambda\beta$

- 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.
  By the linearity of differentiation, we have that for all $j = 1, \ldots, d$,

$$\begin{aligned}
\frac{\partial F}{\partial \beta_j} &= \frac{\partial}{\partial \beta_j}\left\{\frac{1}{n}\sum_{i=1}^{n}(y_i - x_i^T\beta)^2 \; + \lambda\|\beta\|_2^2\right\} \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial \beta_j}(y_i - x_i^T\beta)^2 + \frac{\partial}{\partial \beta_j}\lambda\|\beta\|_2^2 \\
&= -\frac{2}{n}\sum_{i=1}^{n} x_{ij}(y_i - x_i^T\beta) + 2\lambda\beta_j.
\end{aligned}$$

  Hence, by stacking the partial derivatives in a vector, we get

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

  Alternatively, we can write the objective function as

$$F(\beta) = \frac{1}{n}\langle y - X\beta, y - X\beta\rangle + \lambda\|\beta\|_2^2,$$

  where X is the matrix of x_i's: $X = \begin{bmatrix} x_1^T \\ \vdots \\ x_n^T \end{bmatrix}$. In this case if we differentiate with respect to β, we find

$$\begin{aligned}
\nabla F(\beta) &= \frac{1}{n}\left[\langle -X, y - X\beta\rangle + \langle y - X\beta, -X\rangle\right] + 2\lambda\beta \\
&= -\frac{2}{n}\langle X, y - X\beta\rangle + 2\lambda\beta \\
&= -\frac{2}{n}X^T(y - X\beta) + 2\lambda\beta.
\end{aligned}$$

- Consider the Hitters dataset, which you should load and divide into training and test sets using the code below.[2]

[2]You may encounter problems with the quotes when copying and pasting it. If so, delete the quotes that are there and retype the quotes.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import preprocessing

# Load the data
hitters = pd.read_csv('https://raw.githubusercontent.com/selva86/datasets/'
      'master/Hitters.csv', sep=',', header=0)
hitters = hitters.dropna()

# Create our X matrix with the predictors and y vector with the response
X = hitters.drop('Salary', axis=1)
X = pd.get_dummies(X, drop_first=True)
y = hitters.Salary

# Divide the data into training and test sets. By default, 25% goes into the test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Standardize the data. Note that you can convert a data frame into an array by using `np.array()`.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import preprocessing

# Load the data
hitters = pd.read_csv('https://raw.githubusercontent.com/selva86/datasets/mas
hitters = hitters.dropna()
hitters.head(5)  # Display the first 5 rows

# Create our X matrix with the predictors and y vector with the response
X = hitters.drop('Salary', axis=1)
X = pd.get_dummies(X, drop_first=True)
y = hitters.Salary

# Divide the data into training and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y,
        random_state=0)

# Standardize the data
scaler = preprocessing.StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
scaler = preprocessing.StandardScaler().fit(y_train.values.reshape(-1, 1))
y_train = scaler.transform(y_train.values.reshape(-1, 1)).reshape((-1))
y_test = scaler.transform(y_test.values.reshape(-1, 1)).reshape((-1))
```

- Write a function *computegrad* that computes and returns $\nabla F(\beta)$ for any β. Avoid using `for` loops by vectorizing the computation.

```python
def computegrad(beta, lambduh, x=X_train, y=y_train):
    return -2/len(y)*x.T.dot(y-np.dot(x, beta)) + 2*lambduh*beta
```

- Write a function *graddescent* that implements the gradient descent algorithm described in Algorithm 1. The function *graddescent* calls the function *computegrad* as a sub-routine. The function takes as input the initial point, the constant step-size value, and the maximum number of iterations. The stopping criterion is the maximum number of iterations.

```python
def graddescent(beta_init, eta, lambduh, max_iter=1000):
    """
    Run gradient descent with a fixed step size
    Inputs:
      - beta_init: Starting point
      - eta: Step size (a constant)
      - max_iter: Maximum number of iterations to perform
    Output:
      - beta_vals: Matrix of estimated betas at each iteration,
                   with the most recent values in the last row.
    """
    beta = beta_init
    grad_beta = computegrad(beta, lambduh)
    beta_vals = [beta]
    iter_num = 0
    while iter_num < max_iter:
        beta = beta - eta*grad_beta
        beta_vals.append(beta)
        grad_beta = computegrad(beta, lambduh)
        iter_num += 1

    return np.array(beta_vals)
```

- Set the constant step-size to $\eta = 0.05$ and the maximum number of iterations to 1000. Run *graddescent* on the training set of the Hitters dataset for $\lambda = -5.00$. Plot the curve of the objective value $F(\beta_t)$ versus the iteration counter t. Again, avoid using `for` loops when computing the objective values. What do you observe?

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

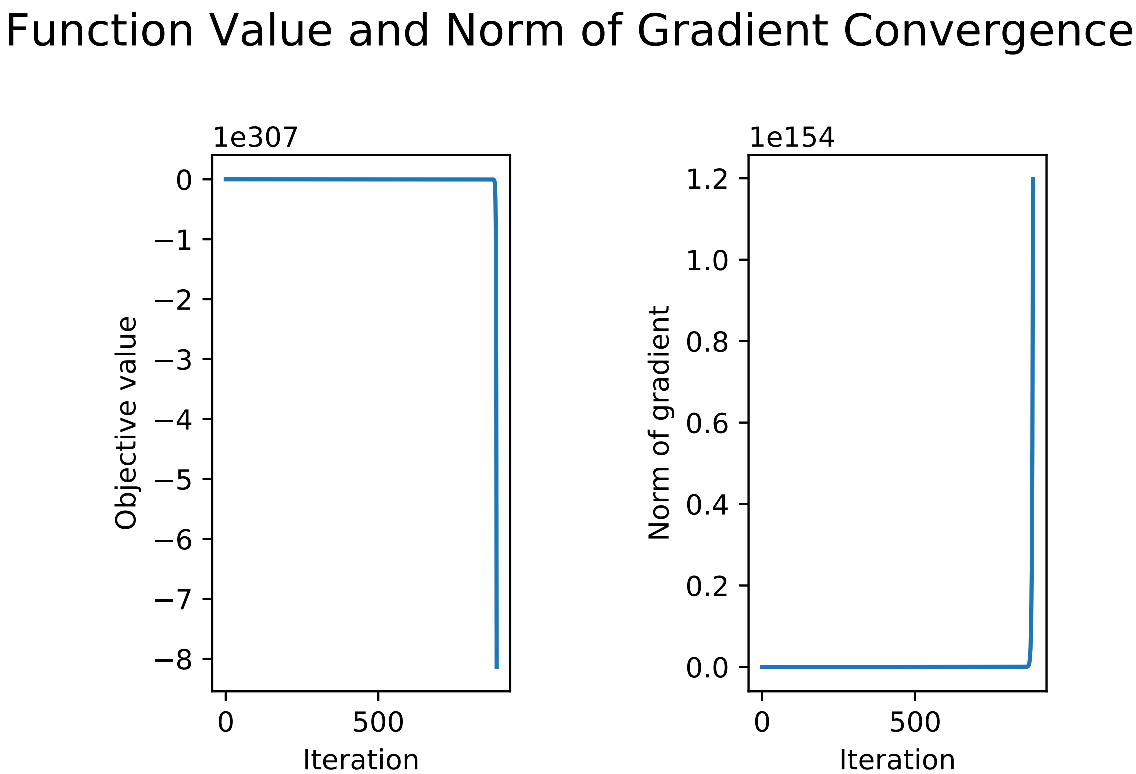
```python
import matplotlib.pyplot as plt

def convergence_plots(x_vals, lambduh):
    """
```

```
    Plot the convergence in terms of the function values and the gradients
    Input:
      - x_vals: Values the gradient descent algorithm stepped to
    """
    n, d = x_vals.shape
    fs = np.zeros(n)
    grads = np.zeros((n, d))
    for i in range(n):
        fs[i] = obj(x_vals[i], lambduh)
        grads[i, :] = computegrad(x_vals[i], lambduh)
    grad_norms = np.linalg.norm(grads, axis=1)
    plt.subplot(121)
    plt.plot(fs)
    plt.xlabel('Iteration')
    plt.ylabel('Objective value')

    plt.subplot(122)
    plt.plot(grad_norms)
    plt.xlabel('Iteration')
    plt.ylabel('Norm of gradient')

    plt.suptitle('Function Value and Norm of Gradient Convergence',\
             fontsize=16)
    plt.subplots_adjust(left=0.2, wspace=0.8, top=0.8)

    plt.show()
```

```
eta = 0.05
max_iter = 1000
lambduh = -5
d = X_train.shape[1]
beta_init = np.zeros(d)
betas = graddescent(beta_init, eta, lambduh, max_iter=max_iter)
convergence_plots(betas, lambduh)
```

```
/home/corinne/anaconda3/envs/pweave/bin/pweave:53: RuntimeWarning:
overflow encountered in double_scalars
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in double_scalars
/home/corinne/anaconda3/envs/pweave/bin/pweave:53: RuntimeWarning:
invalid value encountered in double_scalars
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in square
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2506: RuntimeWarning: overflow
encountered in multiply
  s = (x.conj() * x).real
```

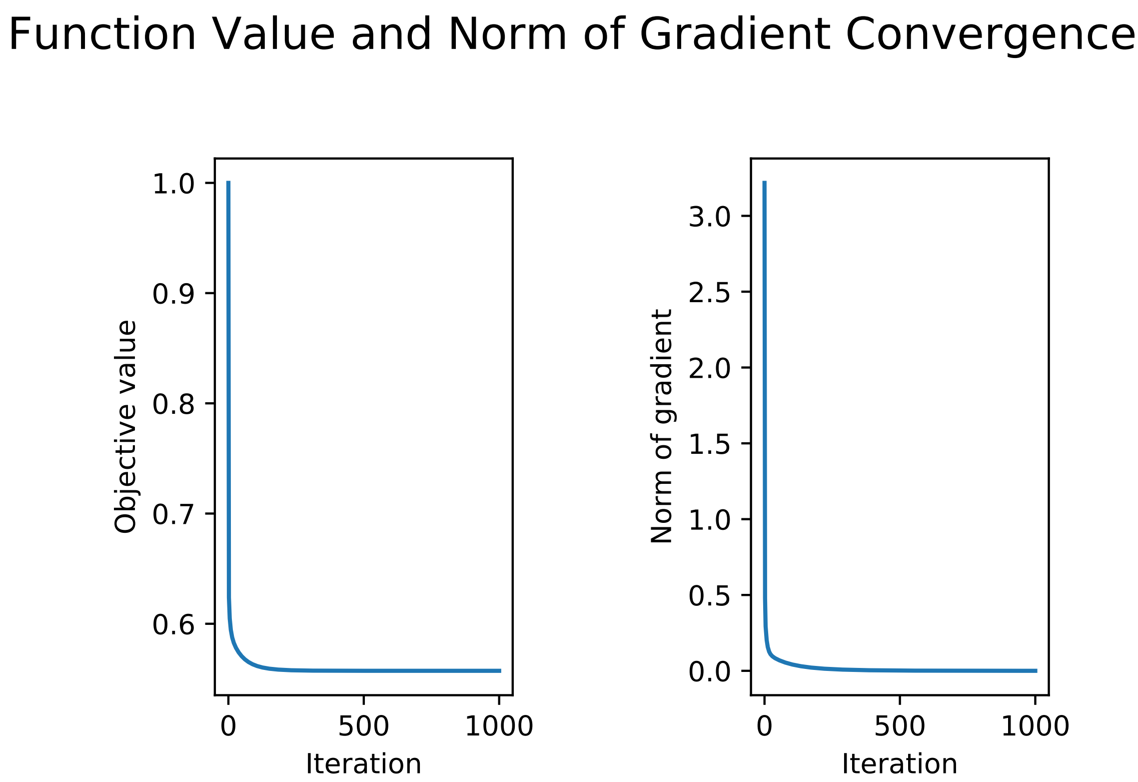
```
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2507: RuntimeWarning: overflow
encountered in reduce
  return sqrt(add.reduce(s, axis=axis, keepdims=keepdims))
```



The objective value and norm of the gradient diverge. With this negative value of λ the objective is unbounded below.

- Set the constant step-size to $\eta = 0.05$ and the maximum number of iterations to 1000. Run *graddescent* on the training set of the Hitters dataset for $\lambda = +0.05$. Plot the curve of the objective value $F(\beta_t)$ versus the iteration counter t. Again, avoid using `for` loops when computing the objective values. What do you observe?

```
eta = 0.05
max_iter = 1000
lambduh = 0.05
d = X_train.shape[1]
beta_init = np.zeros(d)
betas = graddescent(beta_init, eta, lambduh, max_iter=max_iter)
convergence_plots(betas, lambduh)
```



It converges quite quickly to the optimum.

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

  Note that the scikit-learn objective function is

$$\min_{\beta \in \mathbb{R}^d} \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{d} \beta_j x_{ij} \right)^2 + \alpha \sum_{j=1}^{d} \beta_j^2 \,. \tag{6}$$

  (http://scikit-learn.org/stable/modules/linear_model.html#ridge-regressi
  The argmin of this expression is the same as the argmin of

$$\min_{\beta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{d} \beta_j x_{ij} \right)^2 + \frac{\alpha}{n} \sum_{j=1}^{d} \beta_j^2 \,. \tag{7}$$

  Therefore, we have $\lambda = \frac{\alpha}{n}$, i.e., $\alpha = \lambda n$.

```
from sklearn.linear_model import Ridge

n = len(y_train)
alpha = n*lambduh
ridge = Ridge(alpha=alpha, fit_intercept=False)
```

```
ridge.fit(X_train, y_train)
print(ridge.coef_)
print(betas[-1])
```

```
[-0.23105578  0.2868879   0.07468879  0.01210858  0.05255367
0.19391759
 -0.08481254  0.03199856  0.3052965  -0.05615702  0.29085775
0.08474887
 -0.20179687  0.13217803  0.05490176 -0.08577296  0.04089966
-0.12642232
 -0.00546326]
[-0.23098984  0.28676909  0.07465949  0.01219051  0.0525809
0.19389427
 -0.08486348  0.03237805  0.30515384 -0.05617982  0.29063258
0.08477247
 -0.20176225  0.13217398  0.05488663 -0.08577894  0.04090118
-0.12643012
 -0.00546763]
```

```
print(obj(ridge.coef_, lambduh))
print(obj(betas[-1], lambduh))
```

```
0.557324043118233
0.5573240581070139
```

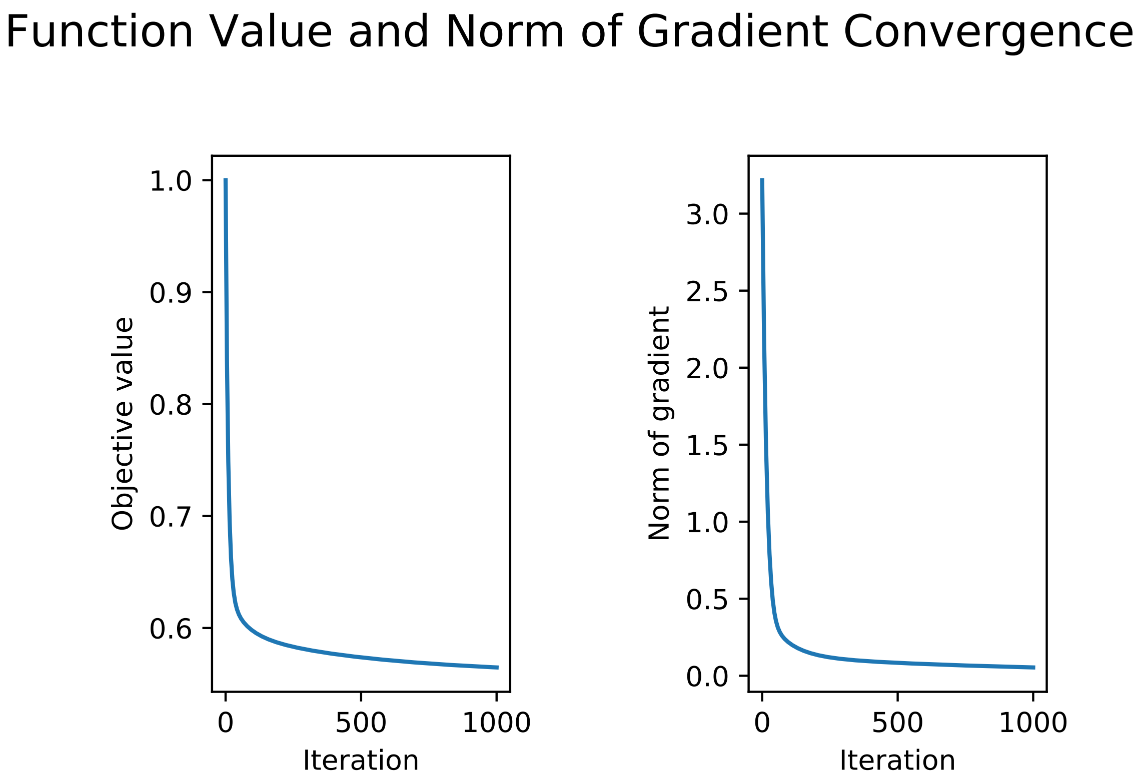
They're the same up to a very high accuracy.

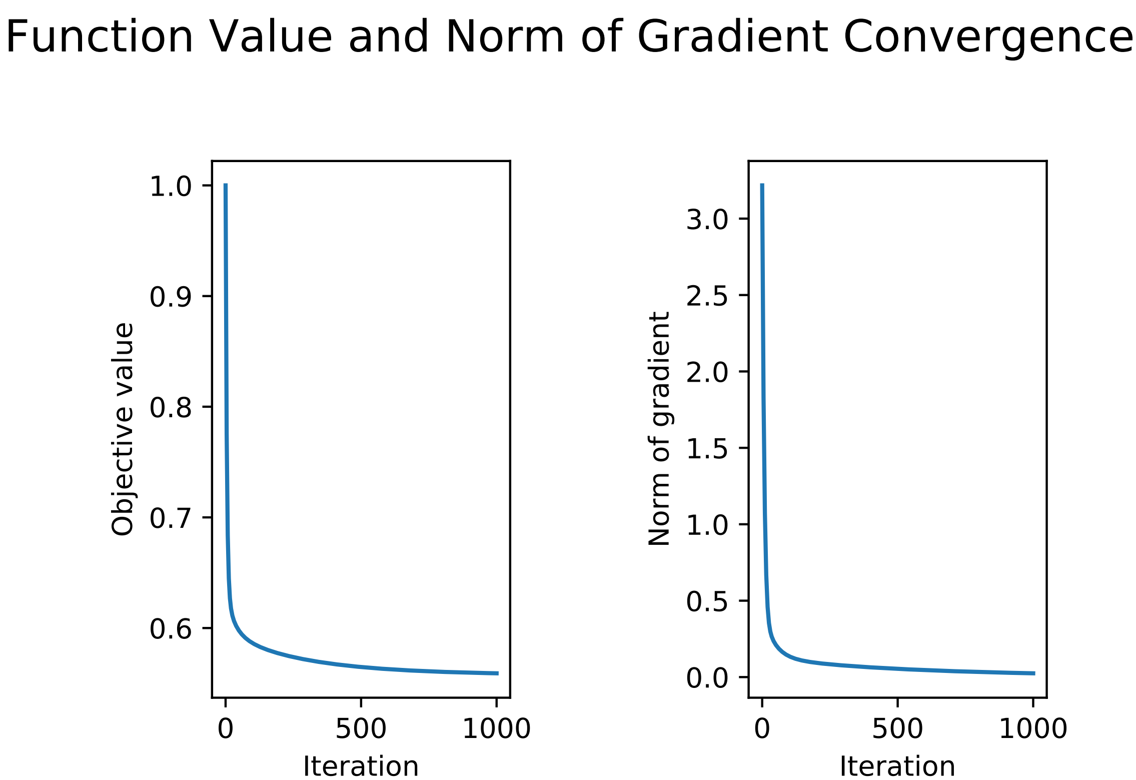
- Run your gradient algorithm for many values of η on a logarithmic scale. Find the final iterate, across all runs for all the values of η, that achieves the smallest value of the objective. Compare β_T to the $\beta^\star$ found by *sklearn.linear_model.Ridge*. Compare the objective value for β_T to the $\beta^\star$. What conclusion to you draw?

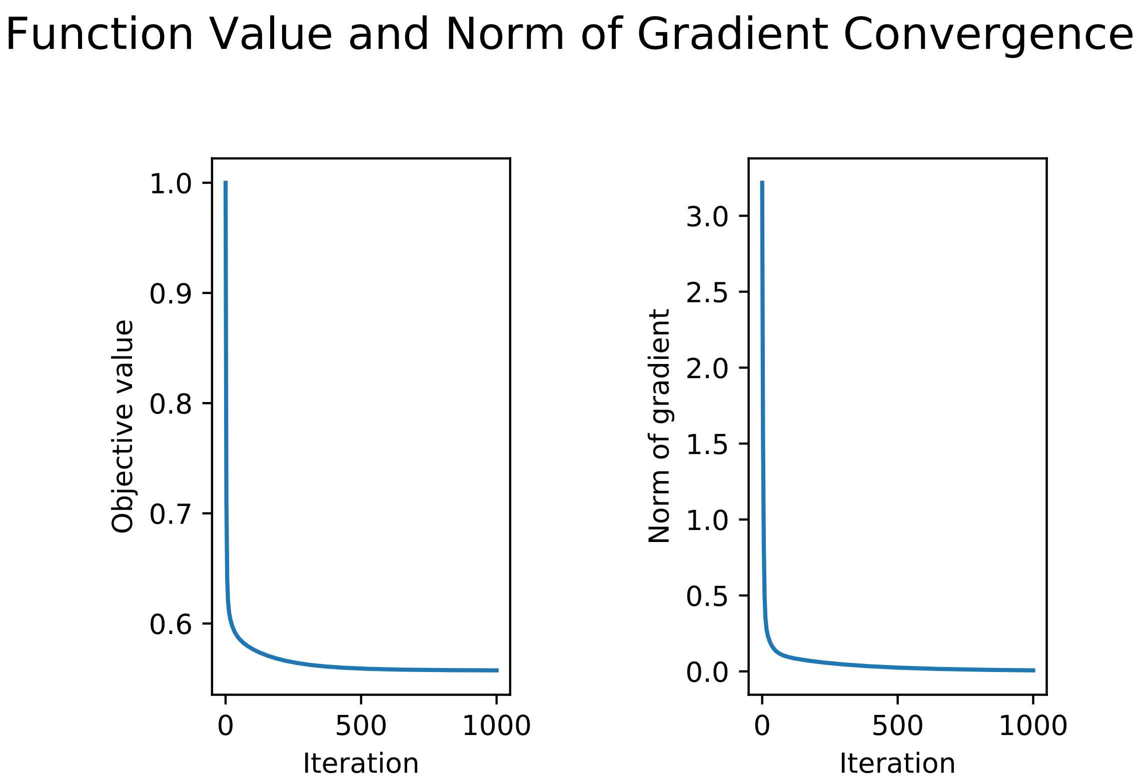
```
obj_vals = []
beta_vals = []
for eta in [2**i for i in range(-8, 0)]:
    print('eta=', eta)
    beta_init = np.zeros(d)
    betas = graddescent(beta_init, eta, lambduh, max_iter=1000)
    convergence_plots(betas, lambduh)
    obj_vals.append(obj(betas[-1], lambduh))
    beta_vals.append(betas[-1])
```

```
eta= 0.00390625
eta= 0.0078125
eta= 0.015625
eta= 0.03125
```

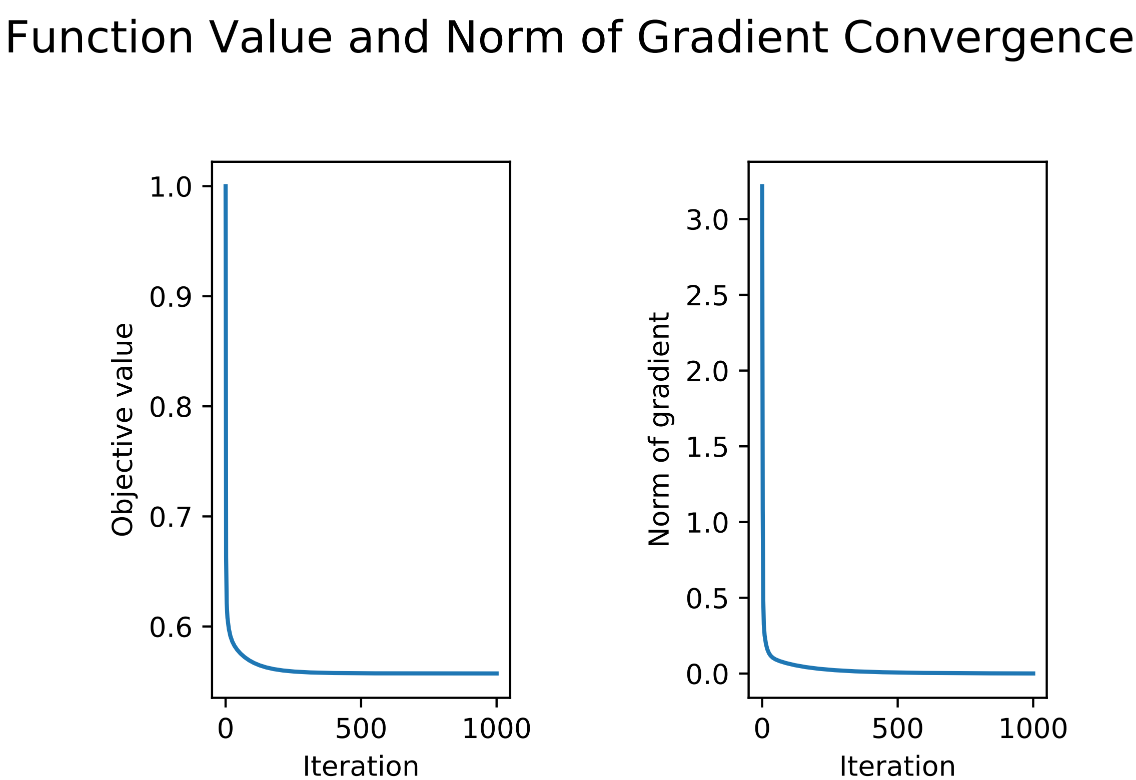
```
eta= 0.0625
eta= 0.125
eta= 0.25
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in double_scalars
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in square
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2506: RuntimeWarning: overflow
encountered in multiply
  s = (x.conj() * x).real
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2507: RuntimeWarning: overflow
encountered in reduce
  return sqrt(add.reduce(s, axis=axis, keepdims=keepdims))
eta= 0.5
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in double_scalars
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in square
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2506: RuntimeWarning: overflow
encountered in multiply
  s = (x.conj() * x).real
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2507: RuntimeWarning: overflow
encountered in reduce
  return sqrt(add.reduce(s, axis=axis, keepdims=keepdims))
```


Function Value and Norm of Gradient Convergence
Objective value
Iteration
1.0
0.9
0.8
0.7
0.6
0
500
1000
Norm of gradient
Iteration
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
500
1000

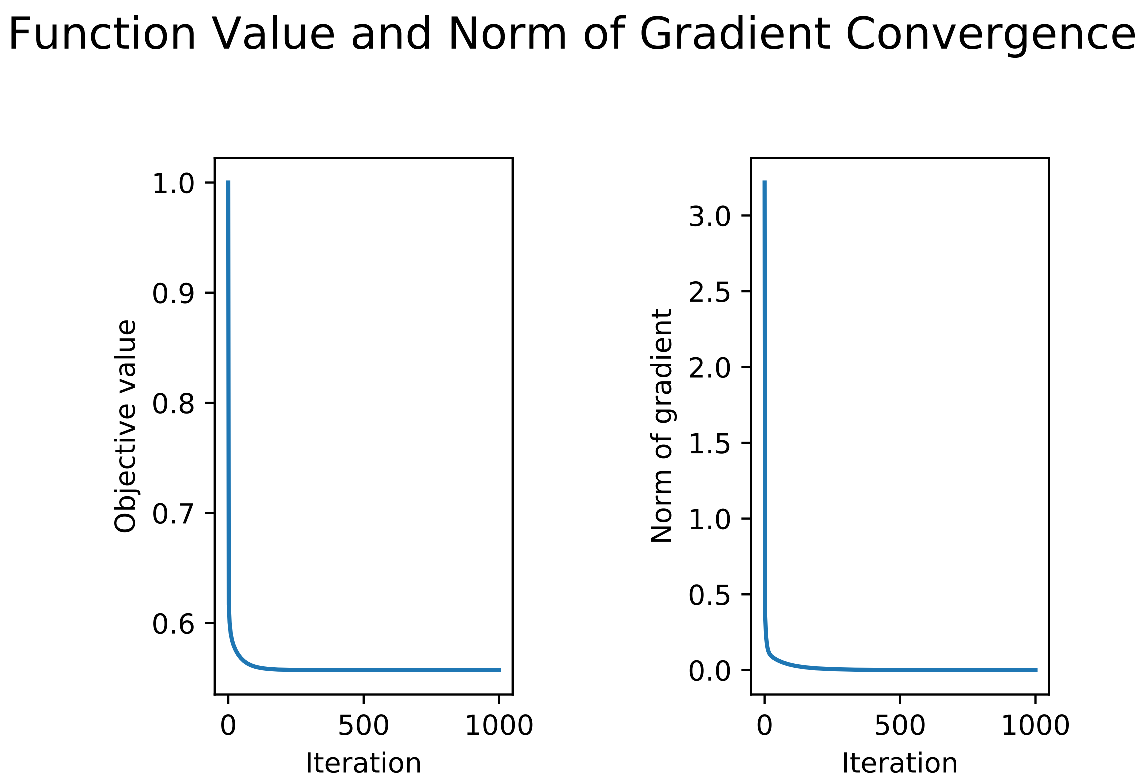

Function Value and Norm of Gradient Convergence
Objective value
Iteration
1.0
0.9
0.8
0.7
0.6
0
500
1000
Norm of gradient
Iteration
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
500
1000

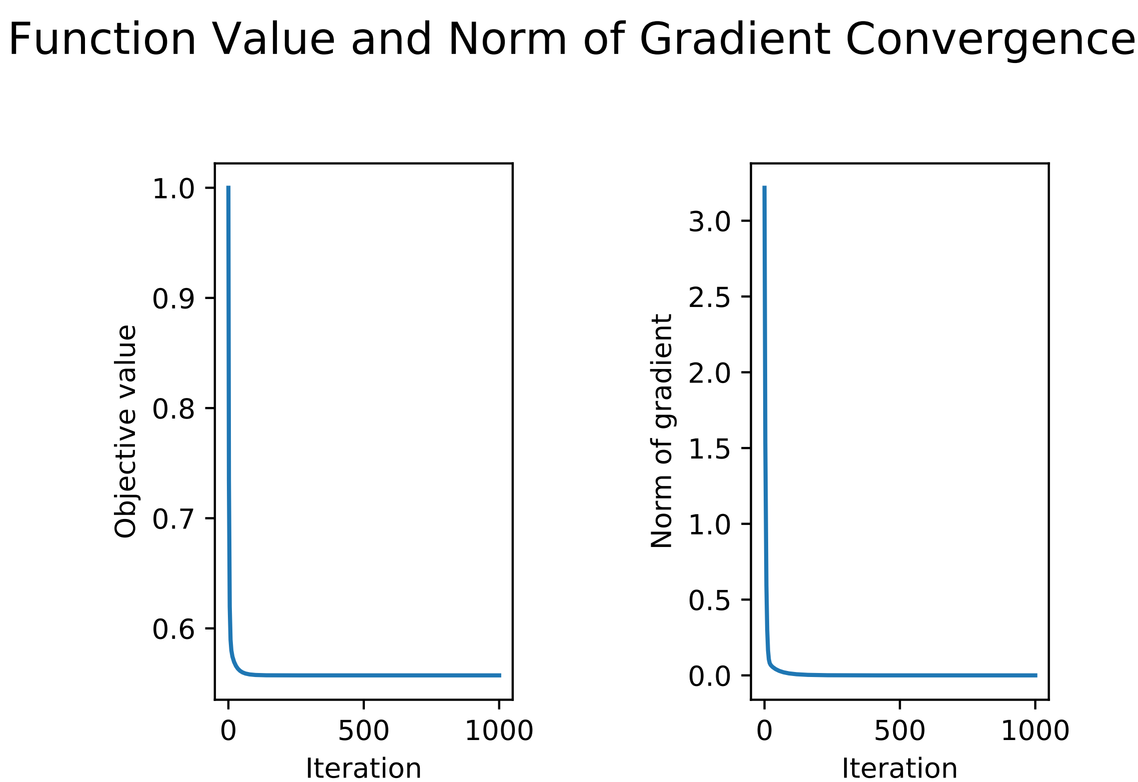


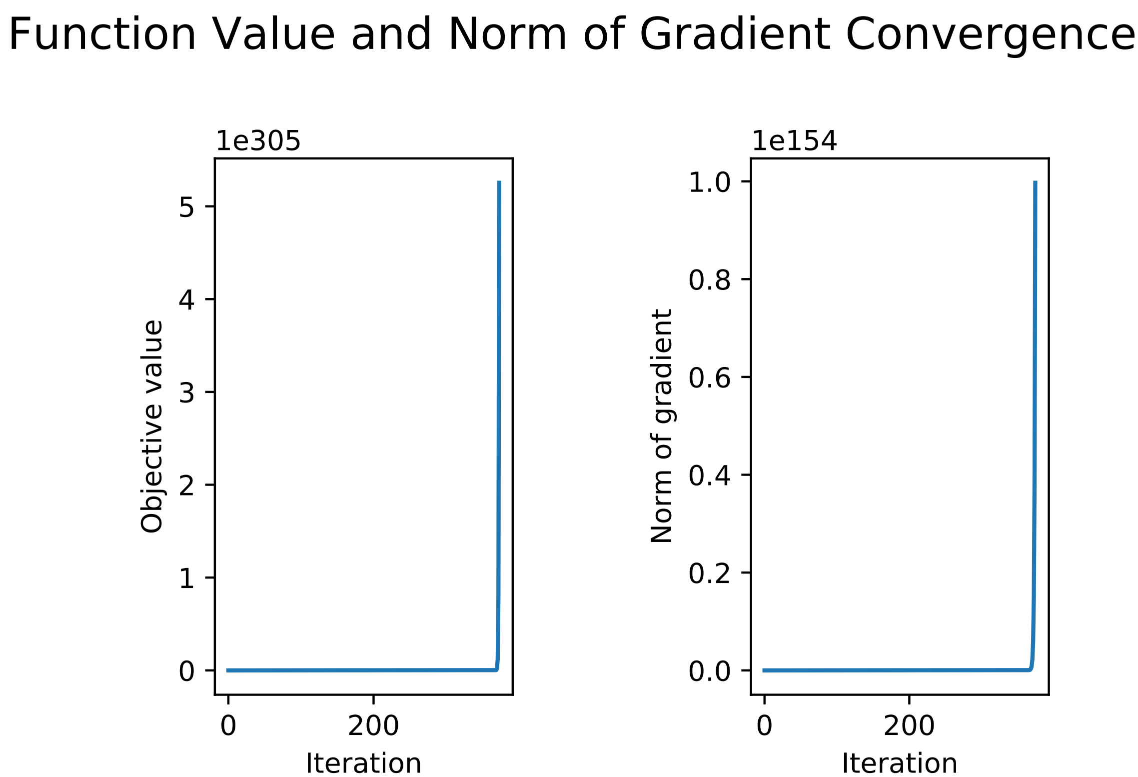
Function Value and Norm of Gradient Convergence
Objective value
1.0
0.9
0.8
0.7
0.6
0
500
1000
Iteration
Norm of gradient
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
500
1000
Iteration

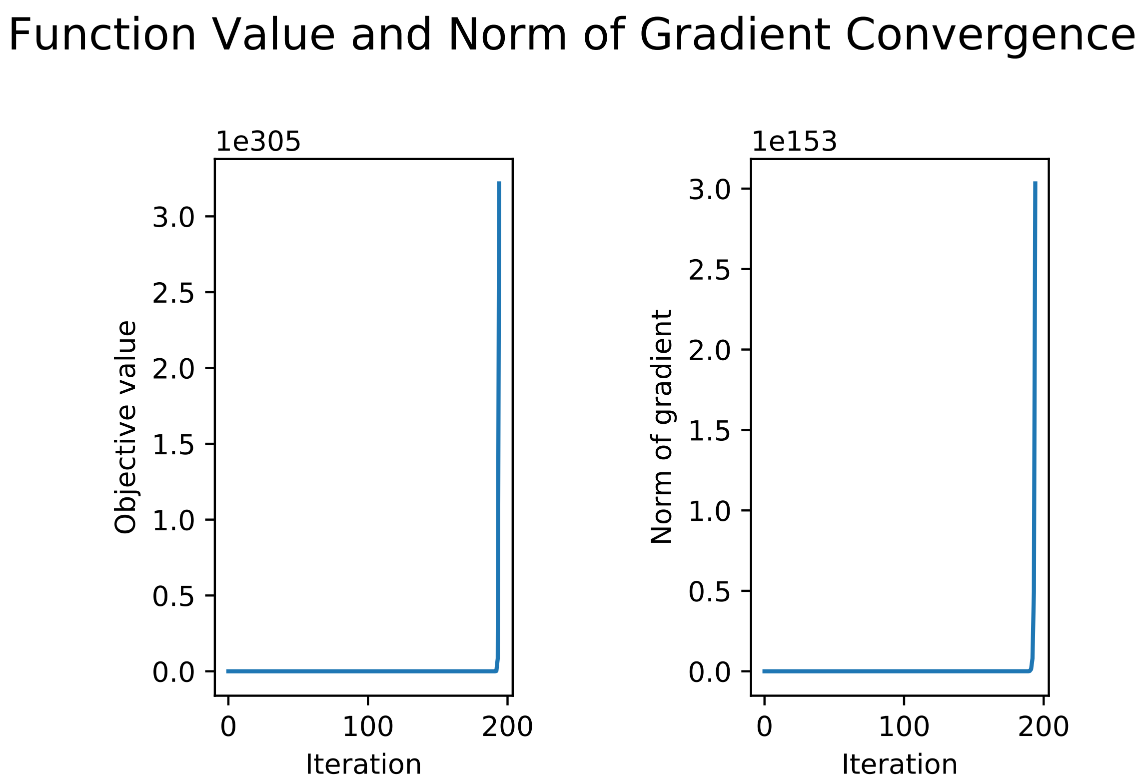


Function Value and Norm of Gradient Convergence
Objective value
1.0
0.9
0.8
0.7
0.6
0
500
1000
Iteration
Norm of gradient
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
500
1000
Iteration


Function Value and Norm of Gradient Convergence
Objective value
Iteration
Norm of gradient
Iteration


Function Value and Norm of Gradient Convergence
Objective value
Iteration
Norm of gradient
Iteration






```
best_obj_idx = np.nanargmin(obj_vals)
print(obj_vals[best_obj_idx])
```

```python
print(obj(ridge.coef_, lambduh))

print(ridge.coef_)
print(beta_vals[best_obj_idx])
```

```
0.5573240431182338
0.557324043118233
[-0.23105578  0.2868879   0.07468879  0.01210858  0.05255367
0.19391759
 -0.08481254  0.03199856  0.3052965  -0.05615702  0.29085775
0.08474887
 -0.20179687  0.13217803  0.05490176 -0.08577296  0.04089966
-0.12642232
 -0.00546326]
[-0.23105579  0.28688791  0.07468878  0.01210858  0.05255367
0.1939176
 -0.08481255  0.03199864  0.30529644 -0.05615702  0.29085774
0.08474886
 -0.20179688  0.13217803  0.05490176 -0.08577296  0.04089965
-0.12642232
 -0.00546326]
```

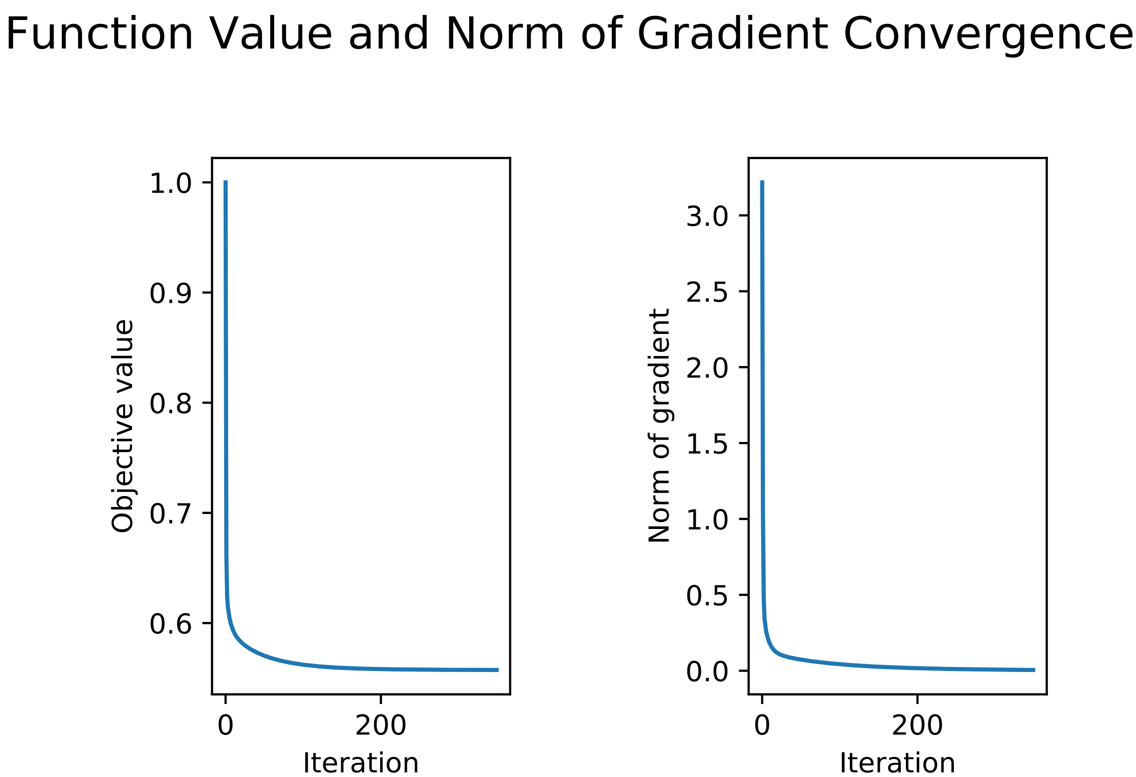
The values are still pretty much the same. The best step size is close to 0.125.

- Change the stopping criterion from being a maximum number of iterations to an ε-stationarity condition $\|\nabla F(\beta)\| \leq \varepsilon$. Redo the last three questions now with this stopping criterion with $\varepsilon = 0.005$. Report your observations.

```python
def graddescent(beta_init, eta, lambduh, epsilon=0.005):
    """
    Run gradient descent with a fixed step size
    Inputs:
      - beta_init: Starting point
      - eta: Step size (a constant)
      - epsilon: Stopping criterion on the norm of the gradient
    Output:
      - beta_vals: Matrix of estimated betas at each iteration,
                   with the most recent values in the last row.
    """
    beta = beta_init
    grad_beta = computegrad(beta, lambduh)
    beta_vals = [beta]
    while np.linalg.norm(grad_beta) > epsilon:
        beta = beta - eta*grad_beta
        beta_vals.append(beta)
        grad_beta = computegrad(beta, lambduh)
```

```
    return np.array(beta_vals)
```

```
eta = 0.05
epsilon = 0.005
lambduh = 0.05
d = X_train.shape[1]
beta_init = np.zeros(d)
betas = graddescent(beta_init, eta, lambduh, epsilon=epsilon)
convergence_plots(betas, lambduh)
```



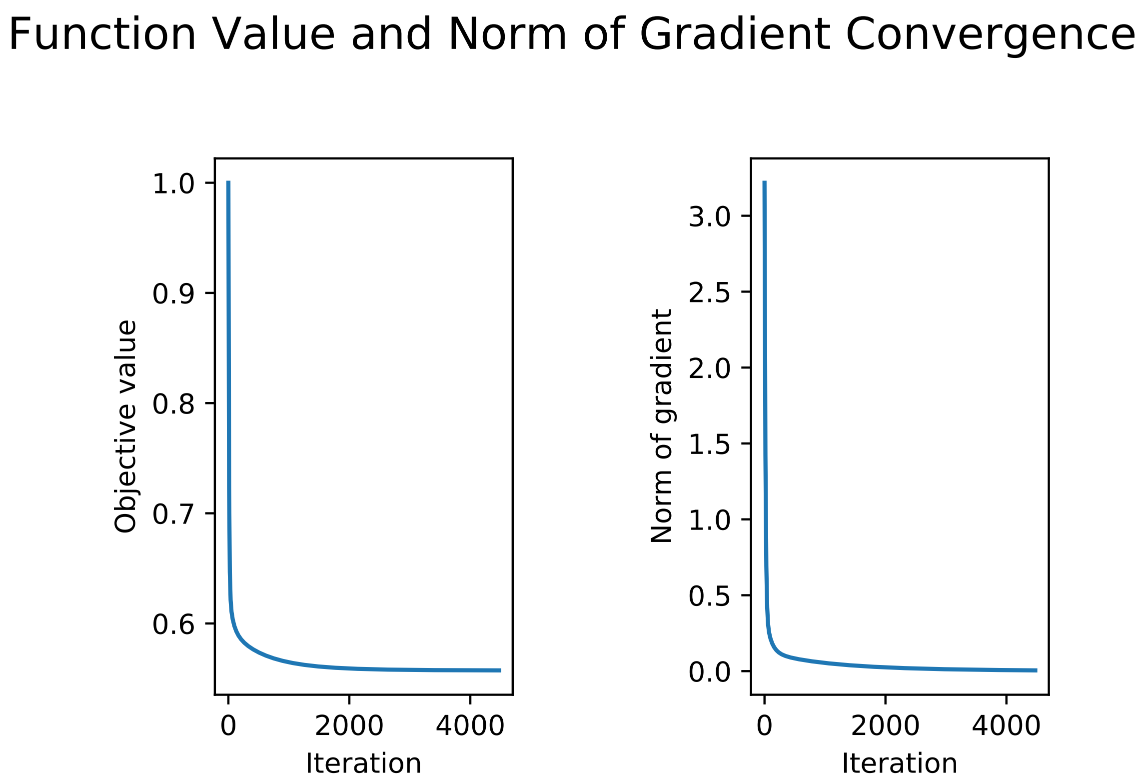

```
print(obj(ridge.coef_, lambduh))
print(obj(betas[-1], lambduh))
```

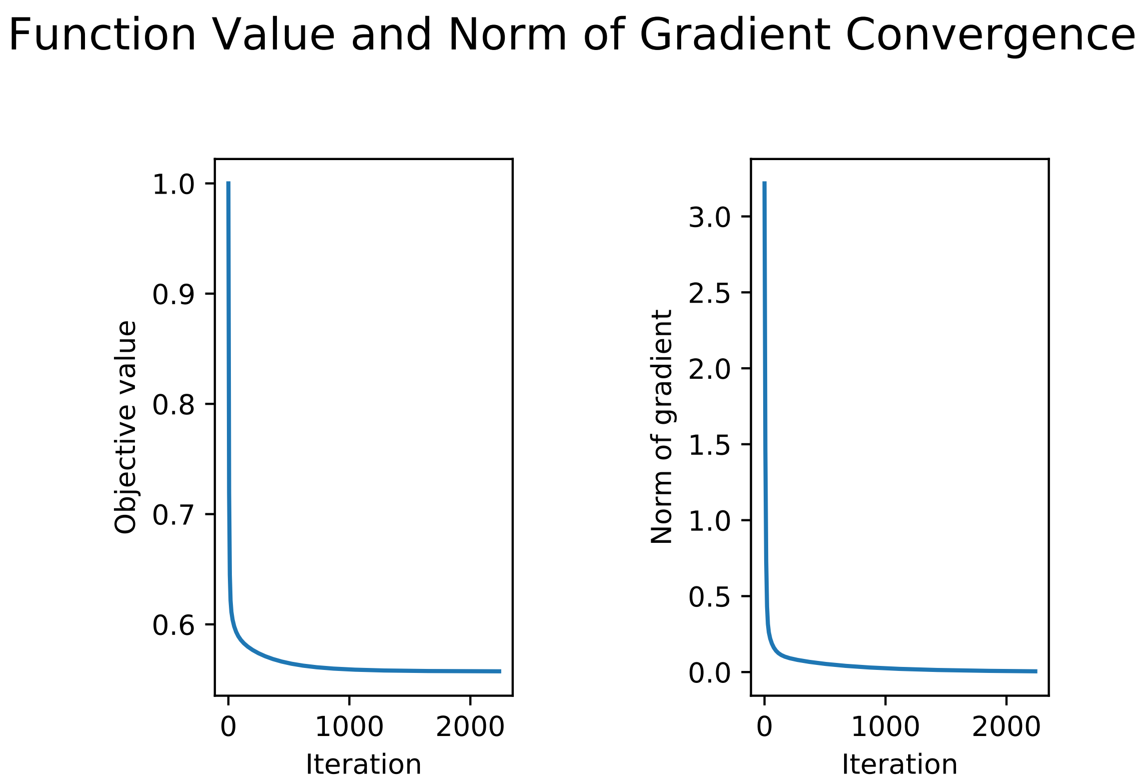
```
0.557324043118233
0.5574095549179856
```

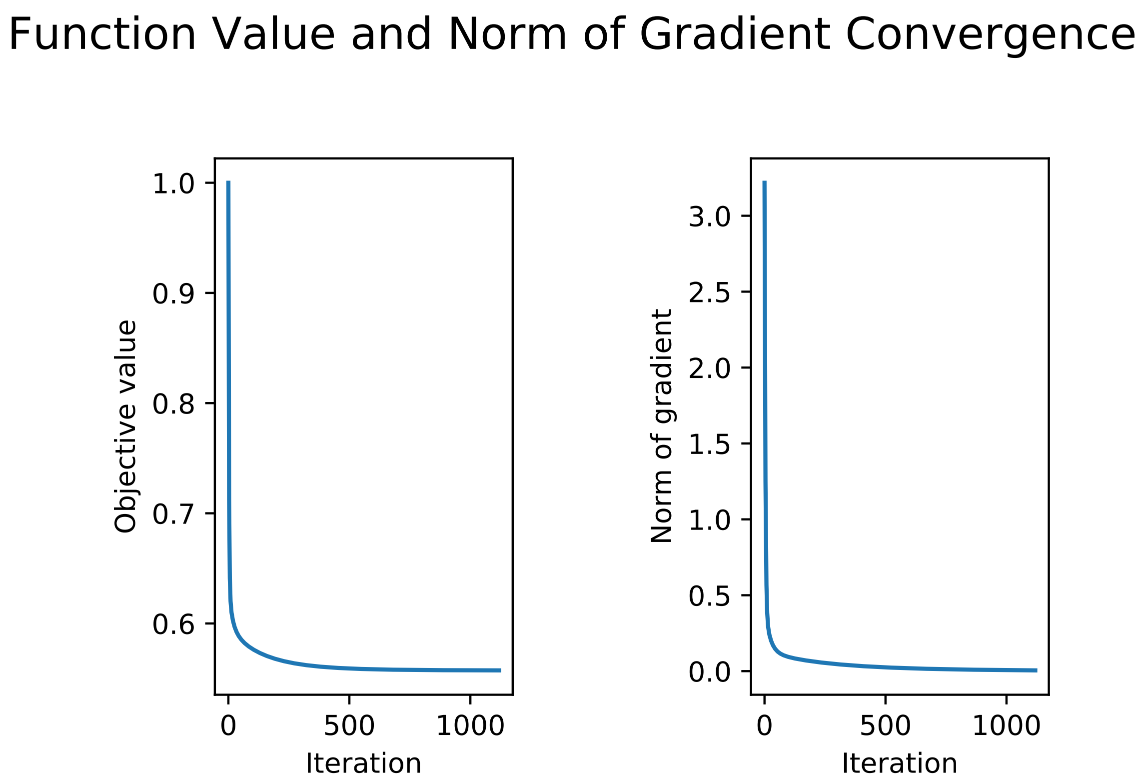
```
obj_vals = []
beta_vals = []
for eta in [2**i for i in range(-8, 0)]:
    print('eta=', eta)
    beta_init = np.zeros(d)
    betas = graddescent(beta_init, eta, lambduh, epsilon=epsilon)
```

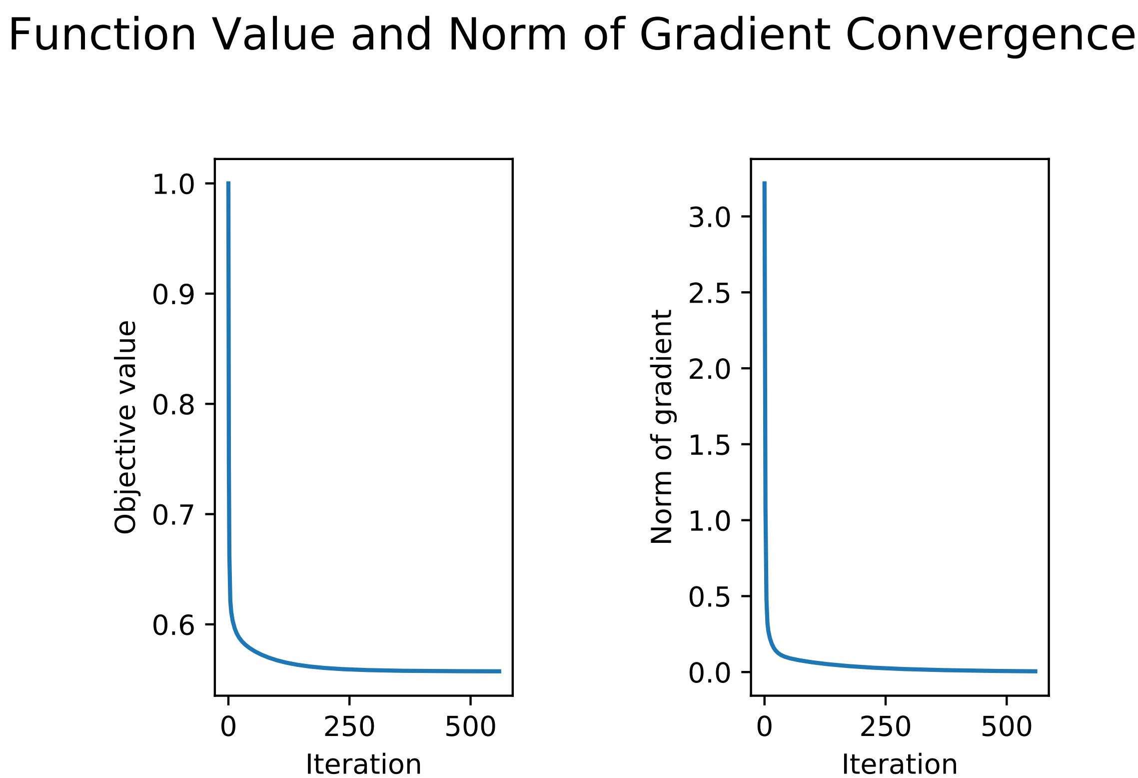
```
    convergence_plots(betas, lambduh)
    obj_vals.append(obj(betas[-1], lambduh))
    beta_vals.append(betas[-1])
```

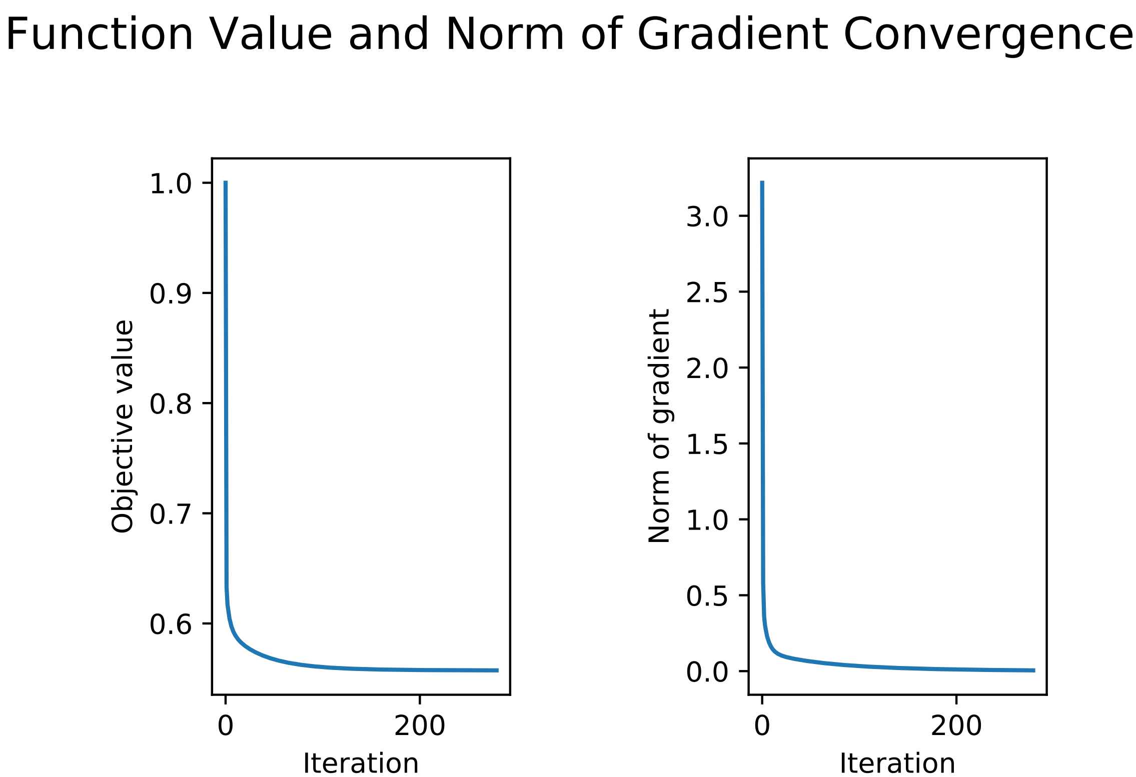
```
eta= 0.00390625
eta= 0.0078125
eta= 0.015625
eta= 0.03125
eta= 0.0625
eta= 0.125
eta= 0.25
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in double_scalars
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in square
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2506: RuntimeWarning: overflow
encountered in multiply
  s = (x.conj() * x).real
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2507: RuntimeWarning: overflow
encountered in reduce
  return sqrt(add.reduce(s, axis=axis, keepdims=keepdims))
eta= 0.5
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in double_scalars
/home/corinne/anaconda3/envs/pweave/bin/pweave:52: RuntimeWarning:
overflow encountered in square
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2506: RuntimeWarning: overflow
encountered in multiply
  s = (x.conj() * x).real
/home/corinne/anaconda3/envs/pweave/lib/python3.6/site-
packages/numpy/linalg/linalg.py:2507: RuntimeWarning: overflow
encountered in reduce
  return sqrt(add.reduce(s, axis=axis, keepdims=keepdims))
```

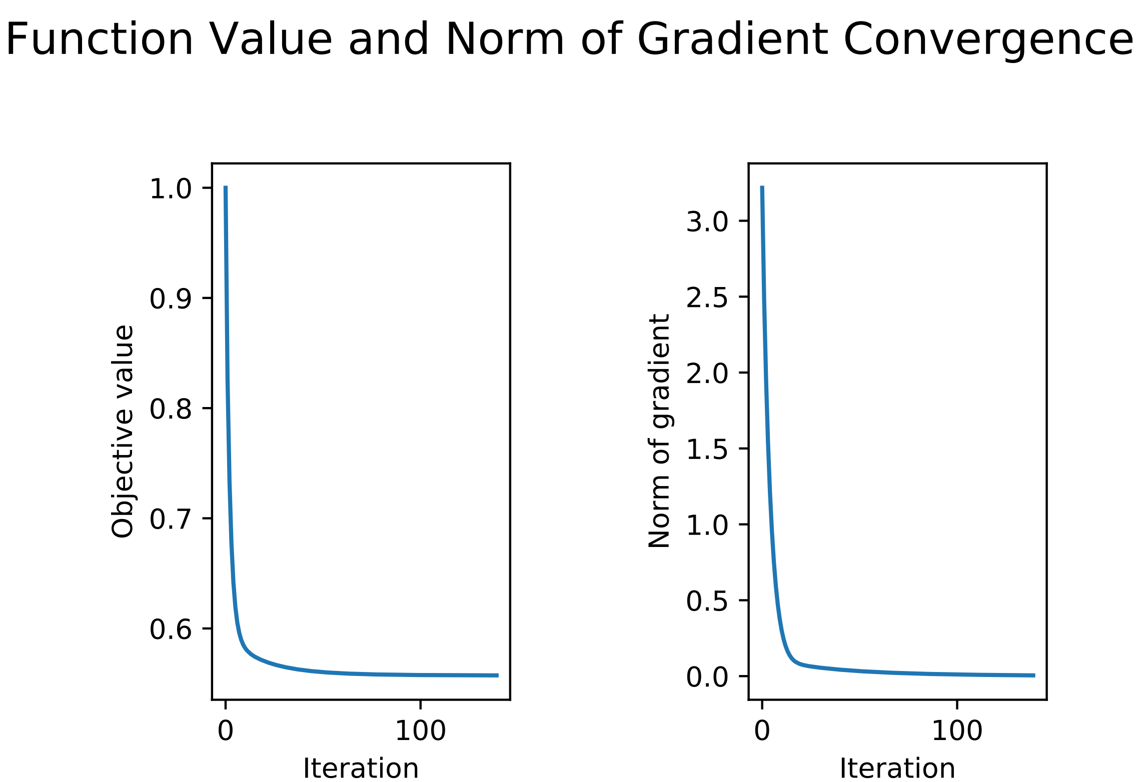

Function Value and Norm of Gradient Convergence
Objective value
Iteration
Norm of gradient
Iteration

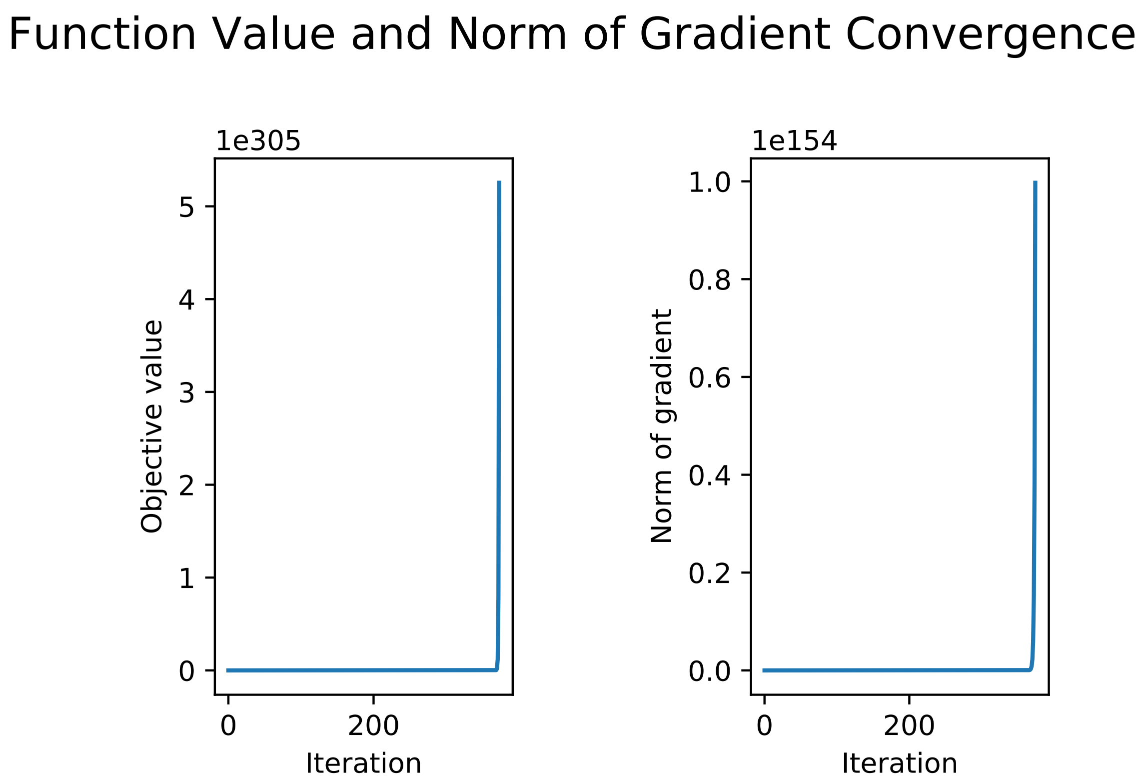

Function Value and Norm of Gradient Convergence
Objective value
Iteration
Norm of gradient
Iteration

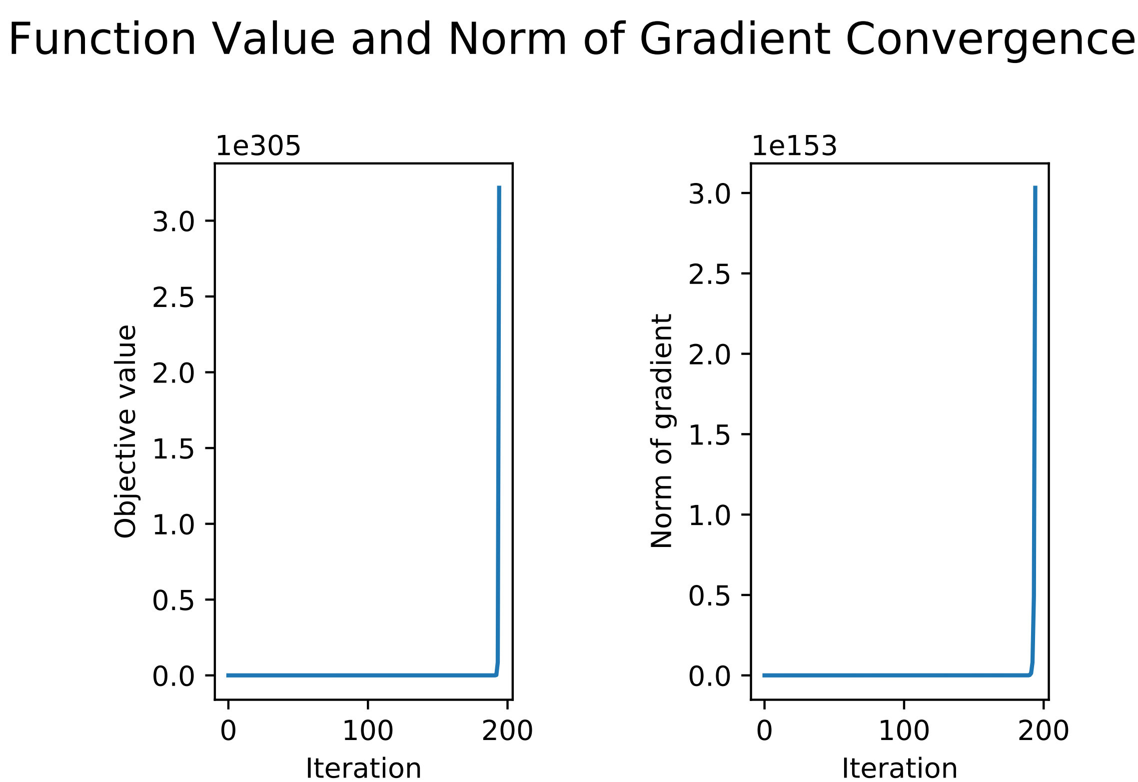

Function Value and Norm of Gradient Convergence
Objective value
1.0
0.9
0.8
0.7
0.6
0
500
1000
Iteration
Norm of gradient
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
500
1000
Iteration


Function Value and Norm of Gradient Convergence
Objective value
1.0
0.9
0.8
0.7
0.6
0
250
500
Iteration
Norm of gradient
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
250
500
Iteration


Function Value and Norm of Gradient Convergence
Objective value
1.0
0.9
0.8
0.7
0.6
0
200
Iteration
Norm of gradient
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
200
Iteration


Function Value and Norm of Gradient Convergence
Objective value
1.0
0.9
0.8
0.7
0.6
0
100
Iteration
Norm of gradient
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
100
Iteration






```
best_obj_idx = np.nanargmin(obj_vals)
print(obj_vals[best_obj_idx])
```

Algorithm 1 Gradient Descent algorithm with fixed constant step-size

input step-size η
initialization $\beta_0 = 0$
repeat for $t = 0, 1, 2, \ldots$
$\beta_{t+1} = \beta_t - \eta \nabla F(\beta_t)$
until the stopping criterion is satisfied.

```
print(obj(ridge.coef_, lambduh))

print(ridge.coef_)
print(beta_vals[best_obj_idx])
```

```
0.5574089597092814
0.557324043118233
[-0.23105578  0.2868879   0.07468879  0.01210858  0.05255367
0.19391759
 -0.08481254  0.03199856  0.3052965  -0.05615702  0.29085775
0.08474887
 -0.20179687  0.13217803  0.05490176 -0.08577296  0.04089966
-0.12642232
 -0.00546326]
[-0.21619563  0.26965163  0.07086261  0.01892748  0.05621002
0.18920536
 -0.08646681  0.04808523  0.2977272  -0.05906319  0.27693413
0.08775234
 -0.19437085  0.13178269  0.05327676 -0.08632454  0.04124321
-0.12680165
 -0.00567509]
```

With this criterion we optimize for fewer iterations. The best step size is unaffected and is still close to 0.125. The estimated coefficients are less close to those of scikit-learn's because we ended further from the optimum.

2 Exercise 2

Exercise 3.8 in Chapter 3 of *An Introduction to Statistical Learning* (in Python):
This question involves the use of simple linear regression on the `Auto` data set.

(a) Read in the dataset. The data can be downloaded from this url: http://faculty.marshall.usc.edu/gareth-james/ISL/Auto.csv When reading in the data use the option `na_values='?'`. Then drop all NaN values using `dropna()`.

```
import pandas as pd
auto = pd.read_csv("http://faculty.marshall.usc.edu/gareth-james/ISL/Auto.csv
```

```
        na_values='?')
print(auto.head(5))
auto = auto.dropna()
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration
year  0  18.0          8         307.0       130.0    3504            12.0
70
1  15.0          8         350.0       165.0    3693            11.5
70
2  18.0          8         318.0       150.0    3436            11.0
70
3  16.0          8         304.0       150.0    3433            12.0
70
4  17.0          8         302.0       140.0    3449            10.5
70

   origin                       name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
3       1              amc rebel sst
4       1                ford torino
```

(b) Use the `OLS` function from the `statsmodels` package to perform a simple linear regression with `mpg` as the response and `weight` as the predictor. Be sure to include an intercept. Use the `summary()` attribute to print the results. Comment on the output. For example:

(i) Is there a relationship between the predictor and the response?

(ii) How strong is the relationship between the predictor and the response?

(iii) Is the relationship between the predictor and the response positive or negative?

Hint: See this URL for help with the statsmodels functions: http://www.statsmodels.org/dev/regression.html#examples

```
import statsmodels.api as sm
import numpy as np

X = auto.iloc[:, 4]
y = auto.iloc[:, 0]

X = sm.add_constant(X)
est = sm.OLS(y, X).fit()
print(est.summary())
```

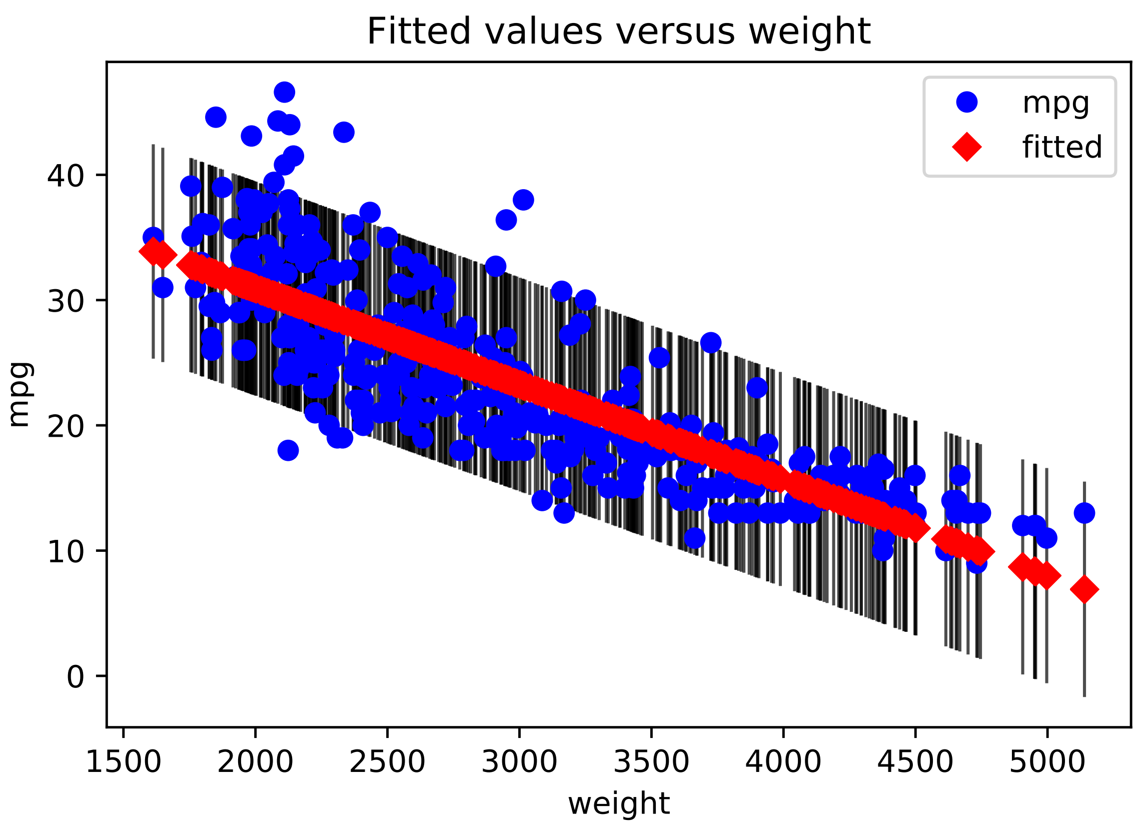
```
                               OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:
0.693
Model:                            OLS   Adj. R-squared:
0.692
Method:                 Least Squares   F-statistic:
878.8
Date:                Mon, 20 Apr 2020   Prob (F-statistic):
6.02e-102
Time:                        12:18:23   Log-Likelihood:
-1130.0
No. Observations:                 392   AIC:
2264.
Df Residuals:                     390   BIC:
2272.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
const         46.2165      0.799     57.867      0.000      44.646
47.787
weight        -0.0076      0.000    -29.645      0.000      -0.008
-0.007
==============================================================================
Omnibus:                       41.682   Durbin-Watson:
0.808
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
60.039
Skew:                           0.727   Prob(JB):
9.18e-14
Kurtosis:                       4.251   Cond. No.
1.13e+04
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 1.13e+04. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

- There is a significant relationship between the predictor and the response, as the p-value for weight is nearly zero.
- The R^2 value is 0.693. Thus, weight explains 69% of the variation in mpg.

- The relationship between the response and the predictor is negative, as the coefficient of weight, -0.0076, is negative. This can also be seen in the scatterplot below.

(c) Plot the response and the predictor using the plot_fit function (http://www.statsmodels.org/dev/generated/statsmodels.graphics.regressionplots.plot_fit.html)

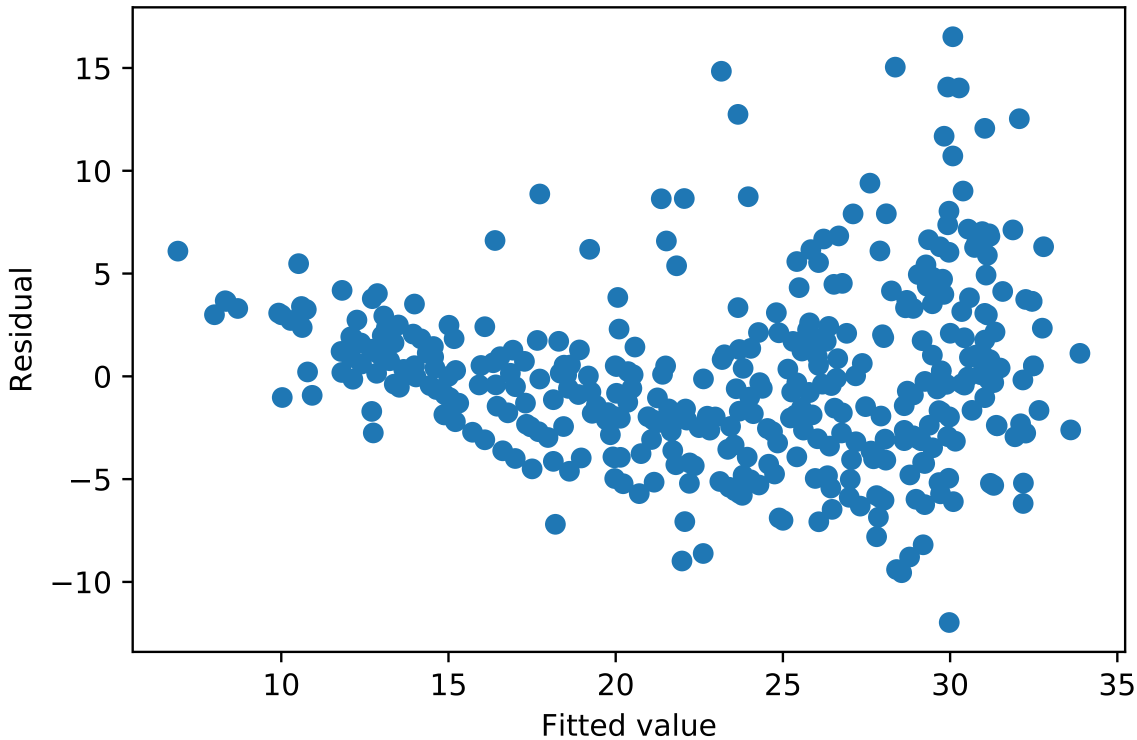
```
sm.graphics.plot_fit(est, 1)
```



Note that the relationship appears to be non-linear.

(d) Plot the residuals vs. fitted values. Comment on any problems you see with the fit.

```
import matplotlib.pyplot as plt
res = est.resid
fitted = est.fittedvalues
plt.clf()
plt.scatter(fitted, res);
plt.xlabel('Fitted value')
plt.ylabel('Residual')
plt.show()
```



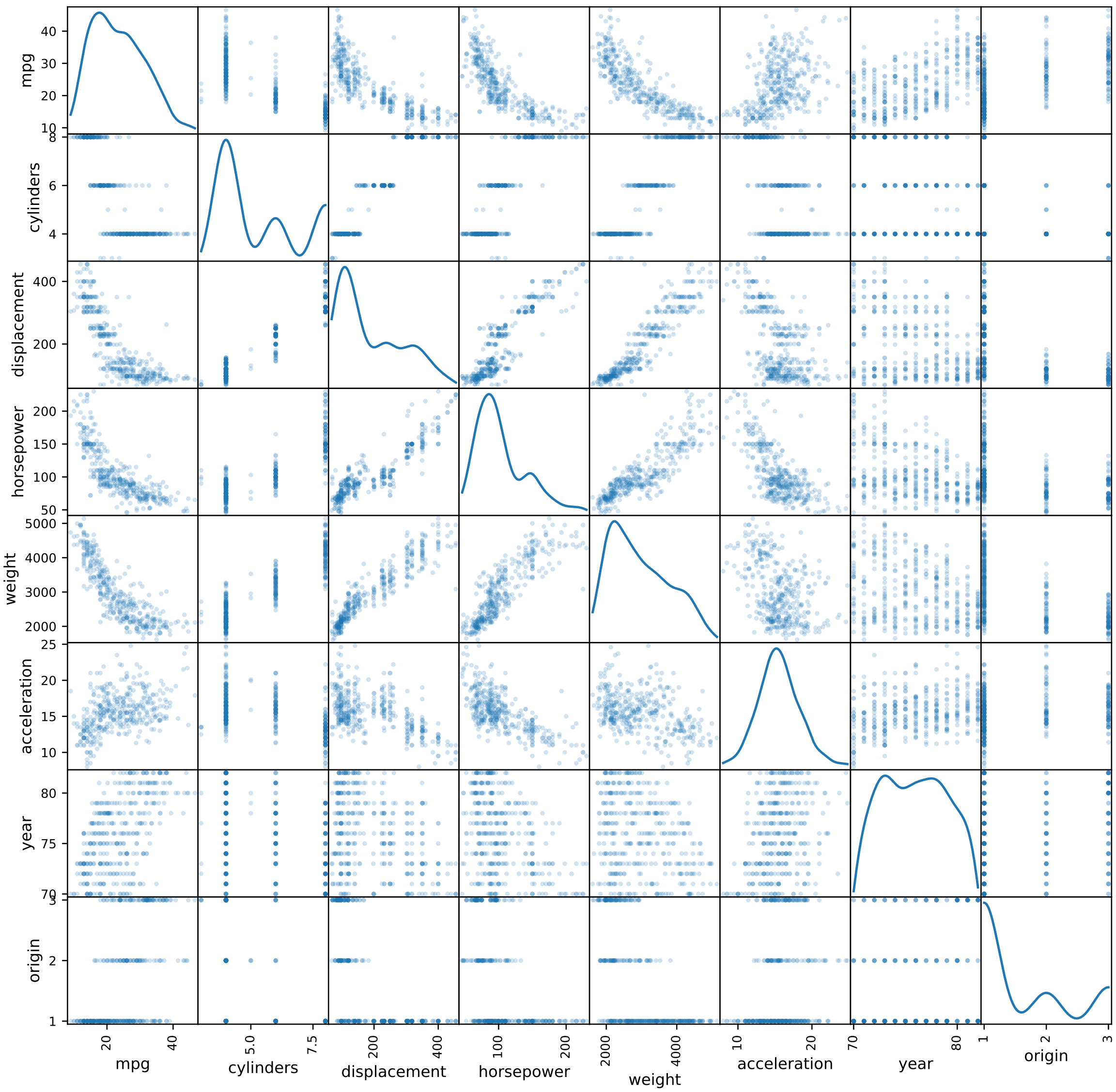
The curvature in the plot of residuals vs. fitted values suggests that a linear model may not be appropriate.

3 Exercise 3

Exercise 3.9 in Chapter 3 of *An Introduction to Statistical Learning* (in Python):
This question involves the use of multiple linear regression on the `Auto` data set.

(a) Produce a scatterplot matrix which includes all of the variables in the data set using `pandas.plotting.scatter_matrix`.

```
import matplotlib.pyplot as plt
%matplotlib inline
from pandas.plotting import scatter_matrix
scatter_matrix(auto, alpha=0.2, figsize=(12, 12), diagonal='kde');
plt.show()
```



(b) Compute the matrix of correlations between the variables using the `corr()` attribute in Pandas.

```
auto.corr()
```

```
                   mpg  cylinders  displacement  horsepower    weight
 mpg          1.000000  -0.777618     -0.805127   -0.778427 -0.832244
cylinders    -0.777618   1.000000      0.950823    0.842983  0.897527
displacement -0.805127   0.950823      1.000000    0.897257  0.932994
horsepower   -0.778427   0.842983      0.897257    1.000000  0.864538
weight       -0.832244   0.897527      0.932994    0.864538  1.000000
acceleration  0.423329  -0.504683     -0.543800   -0.689196 -0.416839
year          0.580541  -0.345647     -0.369855   -0.416361 -0.309120
origin        0.565209  -0.568932     -0.614535   -0.455171 -0.585005
```

```
              acceleration      year    origin
mpg               0.423329  0.580541  0.565209
cylinders        -0.504683 -0.345647 -0.568932
displacement     -0.543800 -0.369855 -0.614535
horsepower       -0.689196 -0.416361 -0.455171
weight           -0.416839 -0.309120 -0.585005
acceleration      1.000000  0.290316  0.212746
year              0.290316  1.000000  0.181528
origin            0.212746  0.181528  1.000000
```

(c) Use the `OLS` function from the `statsmodels` package to perform a multiple linear regression with `mpg` as the response and all other variables except `name` as the predictors. Be sure to include an intercept. Print the results. Comment on the output. For instance:

(i) Is there a relationship between the predictors and the response?

(ii) Which predictors appear to have a statistically significant relationship to the response?

(iii) What does the coefficient for the `year` variable suggest?

```python
X = pd.get_dummies(auto.iloc[:, 1:8], columns=['origin'], drop_first=True)
y = auto.iloc[:, 0]

X = sm.add_constant(X)
est = sm.OLS(y, X).fit()
print(est.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:
0.824
Model:                            OLS   Adj. R-squared:
0.821
Method:                 Least Squares   F-statistic:
224.5
Date:                Mon, 20 Apr 2020   Prob (F-statistic):
1.79e-139
Time:                        12:18:36   Log-Likelihood:
-1020.5
No. Observations:                 392   AIC:
2059.
Df Residuals:                     383   BIC:
2095.
Df Model:                           8
Covariance Type:            nonrobust
==============================================================================
```

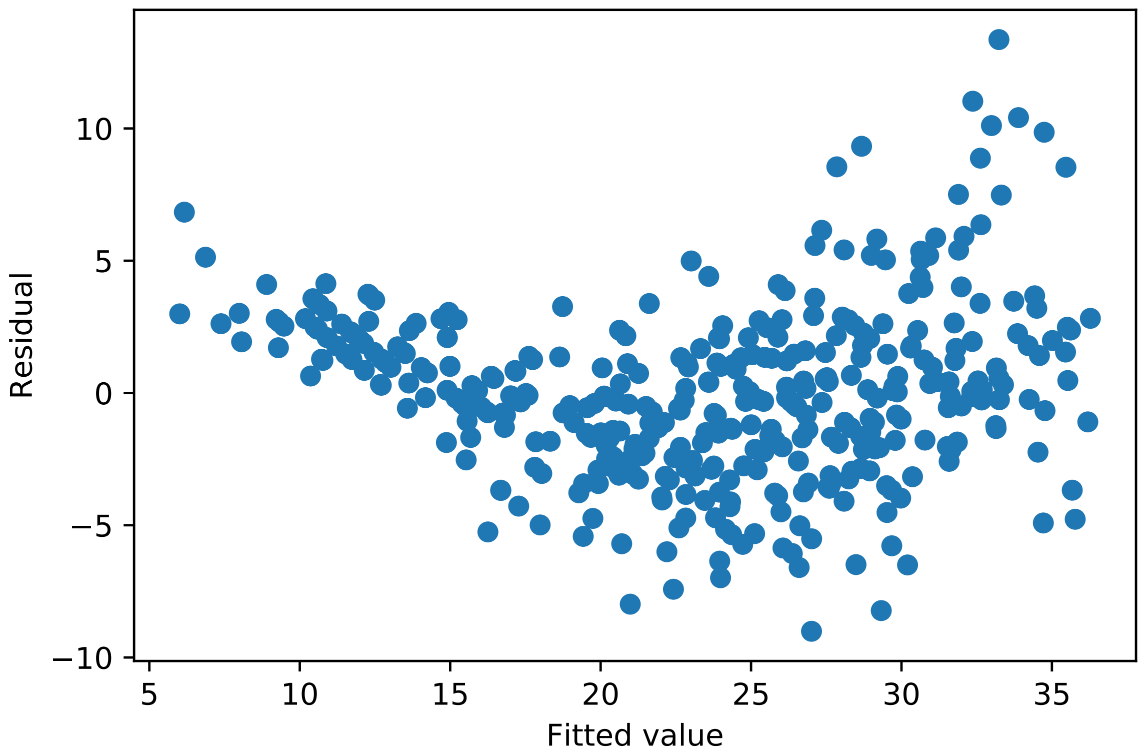
```
                  coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
const         -17.9546      4.677     -3.839      0.000     -27.150
-8.759
cylinders      -0.4897      0.321     -1.524      0.128      -1.121
0.142
displacement    0.0240      0.008      3.133      0.002       0.009
0.039
horsepower     -0.0182      0.014     -1.326      0.185      -0.045
0.009
weight         -0.0067      0.001    -10.243      0.000      -0.008
-0.005
acceleration    0.0791      0.098      0.805      0.421      -0.114
0.272
year            0.7770      0.052     15.005      0.000       0.675
0.879
origin_2        2.6300      0.566      4.643      0.000       1.516
3.744
origin_3        2.8532      0.553      5.162      0.000       1.766
3.940
==============================================================================
Omnibus:                       23.395   Durbin-Watson:
1.291
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
34.452
Skew:                           0.444   Prob(JB):
3.30e-08
Kurtosis:                       4.150   Cond. No.
8.70e+04
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 8.7e+04. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

- Since the p-value of the F-statistic is close to zero, there is a significant relationship between the predictors and the response.
- The predictors that appear to have a statistically significant relationship with the response are displacement, weight, year, and the two origin indicator variables (based on the p-values).
- The coefficient for the variable year suggests that an increase by one year, holding everything else fixed, is associated with an increase in the mpg by 0.78mpg,

on average.

(d) Plot the residuals vs. fitted values. Comment on any problems you see with the fit.

```
res = est.resid
fitted = est.fittedvalues
plt.scatter(fitted, res);
plt.xlabel('Fitted value')
plt.ylabel('Residual')
plt.show()
```



There is some curvature in the residuals vs. fitted plot, suggesting that this linear model might not be the most appropriate.

(e) Statsmodels allows you to fit models using R-style formulas. See http://www.statsmodels.org/dev/example_formulas.html. Use the $*$ and $:$ symbols to fit linear regression models with interaction effects. Do any interactions appear to be statistically significant?

```
import statsmodels.formula.api as smf
est = smf.ols(formula='mpg~cylinders+horsepower+acceleration+year+ \
        C(origin)+year:C(origin)', data=auto).fit()
print(est.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:
0.785
Model:                            OLS   Adj. R-squared:
0.780
Method:                 Least Squares   F-statistic:
174.3
Date:                Mon, 20 Apr 2020   Prob (F-statistic):
1.29e-122
Time:                        12:18:36   Log-Likelihood:
-1060.3
No. Observations:                 392   AIC:
2139.
Df Residuals:                     383   BIC:
2174.
Df Model:                           8
Covariance Type:            nonrobust
==============================================================================
                          coef    std err          t      P>|t|
[0.025      0.975]
------------------------------------------------------------------------------
Intercept               8.1272      6.301      1.290      0.198
-4.261      20.516
C(origin)[T.2]        -37.8227     11.493     -3.291      0.001
-60.419     -15.226
C(origin)[T.3]        -26.0149     10.487     -2.481      0.014
-46.635      -5.395
cylinders              -1.3621      0.230     -5.916      0.000
-1.815      -0.909
horsepower             -0.0890      0.011     -8.010      0.000
-0.111      -0.067
acceleration           -0.4111      0.095     -4.307      0.000
-0.599      -0.223
year                    0.4923      0.073      6.726      0.000
0.348       0.636
year:C(origin)[T.2]     0.5257      0.151      3.490      0.001
0.230       0.822
year:C(origin)[T.3]     0.3816      0.135      2.825      0.005
0.116       0.647
==============================================================================
Omnibus:                       27.844   Durbin-Watson:
1.286
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
39.188
Skew:                           0.531   Prob(JB):
3.09e-09
```

```
Kurtosis:                           4.128   Cond. No.
9.58e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 9.58e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

In general it's good to add interaction terms based on domain knowledge. E.g., if you think the effect of the origin of the vehicle could have changed based on the year (perhaps laws passed requiring higher gas mileage in a certain country).

The interactions between year and origin are statistically significant.

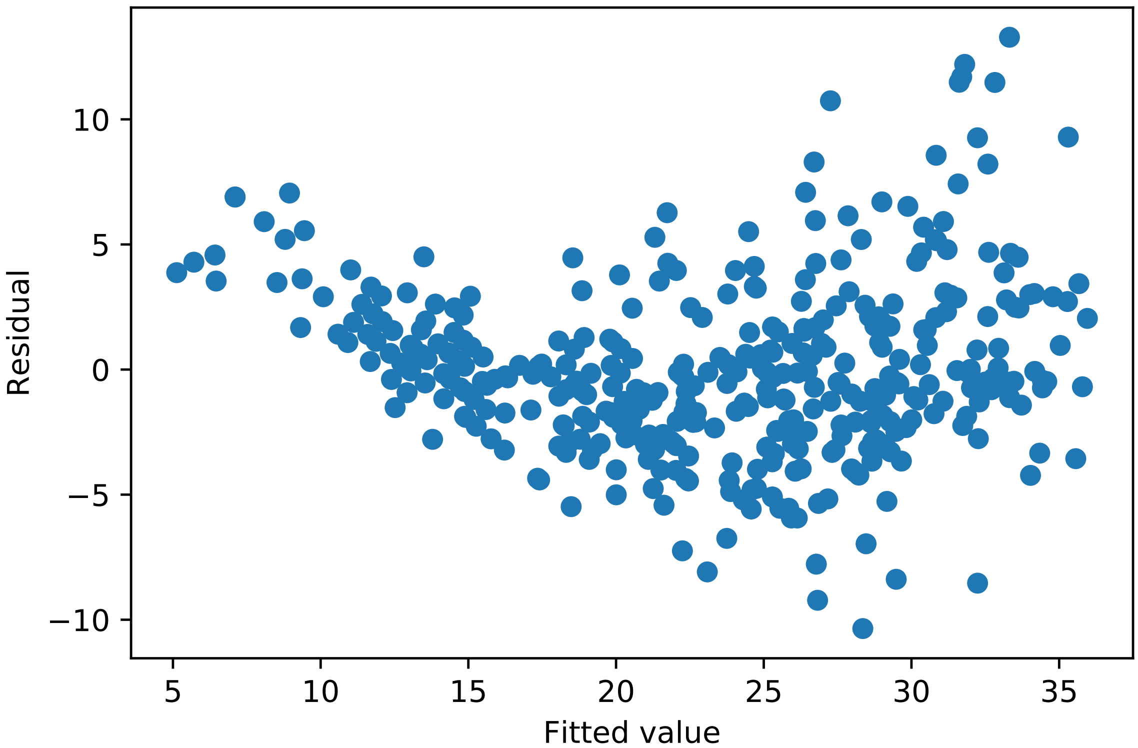
(f) Try a few different transformations of the variables, such as $\log(X)$, $1/X$, $\sqrt{X}$, X^2 . Comment on your findings.
Based on the scatterplots, it looked like we might want to apply a square root or log transformation to displacement, horsepower, and/or weight and possibly also mpg.

```
est = smf.ols(formula='mpg~cylinders+np.sqrt(horsepower)+ \
        acceleration+year+C(origin)', data=auto).fit()
print(est.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:
0.799
Model:                            OLS   Adj. R-squared:
0.795
Method:                 Least Squares   F-statistic:
254.4
Date:                Mon, 20 Apr 2020   Prob (F-statistic):
1.30e-130
Time:                        12:18:36   Log-Likelihood:
-1047.1
No. Observations:                 392   AIC:
2108.
Df Residuals:                     385   BIC:
2136.
Df Model:                           6
Covariance Type:            nonrobust
==============================================================================
                           coef    std err          t      P>|t|
```

```
[0.025      0.975]
----------------------------------------------------------------------------------
Intercept                10.6724      5.441      1.962      0.051
-0.024      21.369
C(origin)[T.2]            2.1739      0.567      3.833      0.000
1.059       3.289
C(origin)[T.3]            3.3971      0.543      6.252      0.000
2.329       4.465
cylinders                -0.7825      0.220     -3.563      0.000
-1.214      -0.351
np.sqrt(horsepower)      -2.5259      0.238    -10.599      0.000
-2.995      -2.057
acceleration             -0.5616      0.093     -6.013      0.000
-0.745      -0.378
year                      0.6600      0.054     12.209      0.000
0.554       0.766
==================================================================================
Omnibus:                        35.776   Durbin-Watson:
1.340
Prob(Omnibus):                   0.000   Jarque-Bera (JB):
55.363
Skew:                            0.611   Prob(JB):
9.51e-13
Kurtosis:                        4.377   Cond. No.
2.40e+03
==================================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 2.4e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

```
res = est.resid
fitted = est.fittedvalues
plt.scatter(fitted, res);
plt.xlabel('Fitted value')
plt.ylabel('Residual')
plt.show()
```

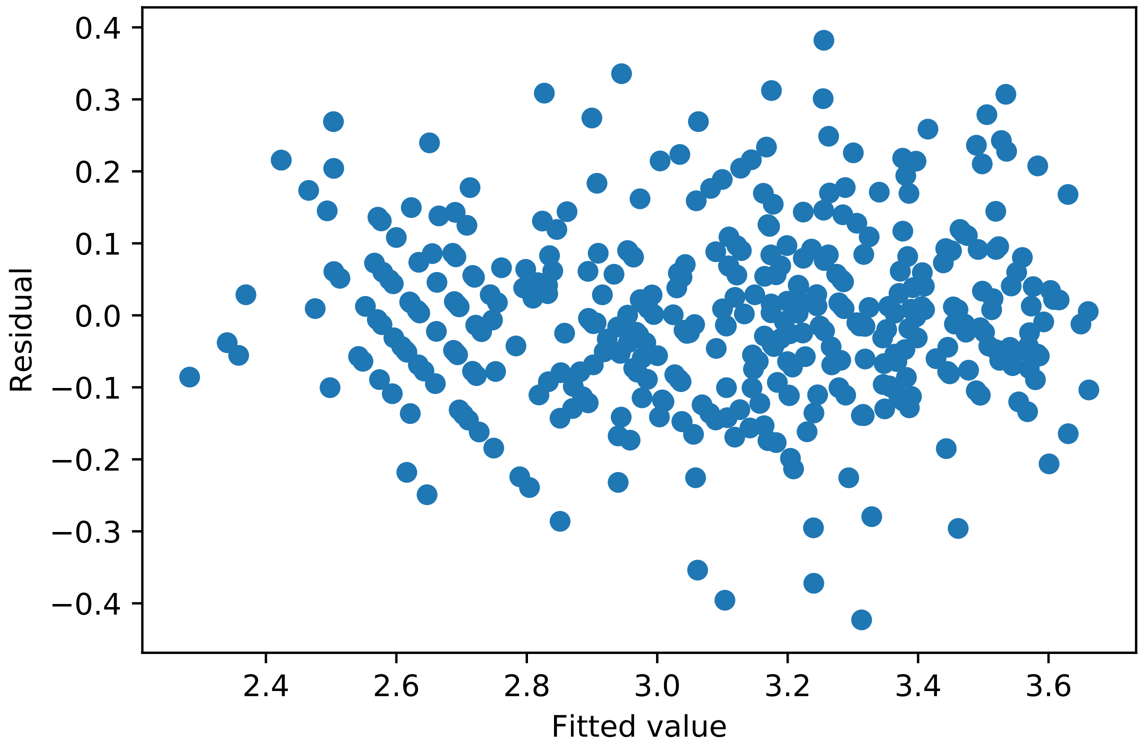



```
est = smf.ols(formula='np.log(mpg)~cylinders+np.log(horsepower)+ \
        acceleration+year+C(origin)', data=auto).fit()
print(est.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:             np.log(mpg)   R-squared:
0.868
Model:                             OLS   Adj. R-squared:
0.866
Method:                  Least Squares   F-statistic:
421.0
Date:                 Mon, 20 Apr 2020   Prob (F-statistic):
1.04e-165
Time:                         12:18:37   Log-Likelihood:
263.65
No. Observations:                  392   AIC:
-513.3
Df Residuals:                      385   BIC:
-485.5
Df Model:                            6
Covariance Type:             nonrobust
==============================================================================
                         coef    std err          t      P>|t|
[0.025      0.975]
```

```
------------------------------------------------------------------------------------
Intercept                 4.6253      0.271     17.042      0.000
4.092       5.159
C(origin)[T.2]            0.0577      0.020      2.879      0.004
0.018       0.097
C(origin)[T.3]            0.0938      0.019      4.874      0.000
0.056       0.132
cylinders                -0.0474      0.008     -6.299      0.000
-0.062      -0.033
np.log(horsepower)       -0.6294      0.043    -14.671      0.000
-0.714      -0.545
acceleration             -0.0279      0.003     -8.439      0.000
-0.034      -0.021
year                      0.0266      0.002     14.023      0.000
0.023       0.030
====================================================================================
Omnibus:                       5.395   Durbin-Watson:
1.523
Prob(Omnibus):                 0.067   Jarque-Bera (JB):
7.234
Skew:                          0.060   Prob(JB):
0.0269
Kurtosis:                      3.655   Cond. No.
3.39e+03
====================================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
[2] The condition number is large, 3.39e+03. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

```
res = est.resid
fitted = est.fittedvalues
plt.scatter(fitted, res);
plt.xlabel('Fitted value')
plt.ylabel('Residual')
plt.show()
```



The residuals vs. fitted plot looks better now.