Regression

May 24, 2021

Warning: Make sure this file is named Regression.ipynb on Coursera or the submit button will not work.

If you plan to run the assignment locally: You can download the assignments and run them locally, but please be aware that as much as we would like our code to be universal, computer platform differences may lead to incorrectly reported errors even on correct solutions. Therefore, we encourage you to validate your solution in Coursera whenever this may be happening. If you decide to run the assignment locally, please: 1. Try to download the necessary data files from your home directory one at a time, 2. Don't update anything other than this Jupyter notebook back to Coursera's servers, and 3. Make sure this notebook maintains its original name after you upload it back to Coursera.

Note: You need to submit the assignment to be graded, and passing the validation button's test does not grade the assignment. The validation button's functionality is exactly the same as running all cells.

[1]:
```
%matplotlib inline
%load_ext autoreload
%autoreload 2

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import pandas as pd

import matplotlib.lines as mlines
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

from aml_utils import test_case_checker
```

1 *Assignment Summary

The following are three problems from the textbook.

Problem 1: At http://www.statsci.org/data/general/brunhild.html, you will find a dataset that measures the concentration of a sulfate in the blood of a baboon as a function of time. Build a linear regression of the log of the concentration against the log of time. * (a) Prepare a plot showing

(a) the data points and (b) the regression line in log-log coordinates. * (b) Prepare a plot showing (a) the data points and (b) the regression curve in the original coordinates. * (c) Plot the residual against the fitted values in log-log and in original coordinates. * (d) Use your plots to explain whether your regression is good or bad and why.

Problem 2: At http://www.statsci.org/data/oz/physical.html, you will find a dataset of measurements by M. Larner, made in 1996. These measurements include body mass, and various diameters. Build a linear regression of predicting the body mass from these diameters. * (a) Plot the residual against the fitted values for your regression. * (b) Now regress the cube root of mass against these diameters. Plot the residual against the fitted values in both these cube root coordinates and in the original coordinates. * (c) Use your plots to explain which regression is better.

Problem 3: At https://archive.ics.uci.edu/ml/datasets/Abalone, you will find a dataset of measurements by W. J. Nash, T. L. Sellers, S. R. Talbot, A. J. Cawthorn and W. B. Ford, made in 1992. These are a variety of measurements of blacklip abalone (Haliotis rubra; delicious by repute) of various ages and genders. * (a) Build a linear regression predicting the age from the measurements, ignoring gender. Plot the residual against the fitted values. * (b) Build a linear regression predicting the age from the measurements, including gender. There are three levels for gender; I'm not sure whether this has to do with abalone biology or difficulty in determining gender. You can represent gender numerically by choosing 1 for one level, 0 for another, and -1 for the third. Plot the residual against the fitted values. * (c) Now build a linear regression predicting the log of age from the measurements, ignoring gender. Plot the residual against the fitted values. * (d) Now build a linear regression predicting the log age from the measurements, including gender, represented as above. Plot the residual against the fitted values. * (e) It turns out that determining the age of an abalone is possible, but difficult (you section the shell, and count rings). Use your plots to explain which regression you would use to replace this procedure, and why. * (f) Can you improve these regressions by using a regularizer? obtain plots of the cross-validated prediction error.

Attention: After finishing this notebook, you will need to do a follow-up quiz as well. The overall grade for this assignment is based on this notebook and the follow-up quiz.

2 Task 1

Write a function `linear_regression` that fits a linear regression model, and takes the following two arguments as input:

1. `X`: A numpy array of the shape `(N,d)` where `N` is the number of data points, and `d` is the data dimension. Do not assume anything about `N` or `d` other than being a positive integer.
2. `Y`: A numpy array of the shape `(N,)` where `N` is the number of data points.
3. `lam`: The regularization coefficient λ, which is a scalar positive value. See the objective function below.

and returns the linear regression weight vector

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \cdots \\ \beta_d \end{bmatrix}$$

which is a numpy array with a shape of `(d+1,1)`. Your function should:

1. **Have an Intercept Weight**: In other words, your fitting model should be minimizing the following mean-squared loss

$$\mathcal{L}(\beta; X, Y, \lambda)^2 = \frac{1}{N}\sum_{i=1}^{N}\left(y^{(i)} - (\beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + \cdots + \beta_d x_d^{(i)})\right)^2 + \lambda\beta^T\beta.$$

   An easy way to do this is by concatenating a constant 1-column to the data matrix (think about the right numpy function and the proper call given the defined loss and weight vector format).

   Hint: The textbook has provided you with the solution for the least squares optimization with ridge regression which could be helpful.

2. **Never Raise An Error, and Return the Solution with the Smallest Euclidean Norm** in case the optimal weight vector is not unique. For instance, when the number of data points is smaller than the dimension, many optimal weight vectors exist.

   Hint: Reviewing your linear algebra may be helpful in this case. You may want to use the Moore-Penrose matrix inversion.

Note: The regularization coefficient will not be used for the first two problems. However, it would be used later, and we expect you to implement it correctly here.

[2]:
```
def linear_regression(X,Y,lam=0):
    """
    Train linear regression model

        Parameters:
                X (np.array): A numpy array with the shape (N, d) where N is␣
↪the number of data points and d is dimension
                Y (np.array): A numpy array with the shape (N,), where N is the␣
↪number of data points
                lam (int): The regularization coefficient where default value␣
↪is 0

        Returns:
                beta (np.array): A numpy array with the shape (d+1, 1) that␣
↪represents the linear regression weight vector
    """
    assert X.ndim==2
    N = X.shape[0]
    d = X.shape[1]


    X_d_1 = np.insert(X, 0, 1, axis=1)


    assert Y.size == N
    xtranspose = X_d_1.T
    xtransx = np.matmul(xtranspose,X_d_1)
```

```
    Y_col = Y.reshape(-1,1)
    lamidentity = np.identity(xtransx.shape[1]) * lam
    matinv = np.linalg.pinv(lamidentity + xtransx)
    xtransy = np.matmul(xtranspose, Y_col)
    beta = np.matmul(matinv, xtransy)




    assert beta.shape == (d+1, 1)
    return beta
```

[3]:
```
some_X = (np.arange(35).reshape(7,5) ** 13) % 20
some_Y = np.sum(some_X, axis=1)
some_beta = linear_regression(some_X, some_Y, lam=0)
assert np.array_equal(some_beta.round(3), np.array([[ 0.],
                                                    [ 1.],
                                                    [ 1.],
                                                    [ 1.],
                                                    [ 1.],
                                                    [ 1.]]))

some_beta_2 = linear_regression(some_X, some_Y, lam=1)
assert np.array_equal(some_beta_2.round(3), np.array([[0.032],
                                                      [0.887],
                                                      [1.08 ],
                                                      [1.035],
                                                      [0.86 ],
                                                      [1.021]]))

another_X = some_X.T
another_Y = np.sum(another_X, axis=1)
another_beta = linear_regression(another_X, another_Y, lam=0)
assert np.array_equal(another_beta.round(3), np.array([[-0.01 ],
                                                       [ 0.995],
                                                       [ 1.096],
                                                       [ 0.993],
                                                       [ 0.996],
                                                       [ 0.995],
                                                       [ 0.946],
                                                       [ 0.966]]))
```

```
      ␣
↪-------------------------------------------------------------------------
```

```
AssertionError                            Traceback (most recent call last)

<ipython-input-3-002167ade8dd> in <module>
 10
 11 some_beta_2 = linear_regression(some_X, some_Y, lam=1)
---> 12 assert np.array_equal(some_beta_2.round(3), np.array([[0.032],
 13                                                          [0.887],
 14                                                          [1.08 ],


AssertionError:
```

[4]:
```
# Checking against the pre-computed test database
test_results = test_case_checker(linear_regression, task_id=1)
assert test_results['passed'], test_results['message']
```

[ ]:
```
# Task 1 Test Cell

# The following are hints to make your life easier duing debugging if you failed the pre-computed tests.
#
#   When an error is raised in checking against the pre-computed test database:
#
#     0. test_results will be a python dictionary, with the bug information stored in it. Don't be afraid to look into it!
#
#     1. You can access the failed test arguments by reading test_results['test_kwargs']. test_results['test_kwargs'] will be
#        another python dictionary with its keys being the argument names and the values being the argument values.
#
#     2. test_results['correct_sol'] will contain the correct solution.
#
#     3. test_results['stu_sol'] will contain your implementation's returned solution.
```

3 Task 2

Write a function `linear_predict` that given the learned weights in the `linear_regression` function predicts the labels. Your functions takes the following two arguments as input:

1. `X`: A numpy array of the shape `(N,d)` where `N` is the number of data points, and `d` is the data dimension. Do not assume anything about `N` or `d` other than being a positive integer.

2. `beta`: A numpy array of the shape `(d+1,1)` where `d` is the data dimension

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \cdots \\ \beta_d \end{bmatrix}$$

Your function should produce the $\hat{y}$ numpy array with the shape of `(N,)`, whose i^{th} element is defined as

$$\hat{y}^{(i)} = \beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + \cdots + \beta_d x_d^{(i)}$$

.

[5]:
```
def linear_predict(X,beta):
    """
    Predict with linear regression model

        Parameters:
                X (np.array): A numpy array with the shape (N, d) where N is␣
 ↪the number of data points and d is dimension
                beta (np.array): A numpy array of the shape (d+1,1) where d is␣
 ↪the data dimension

        Returns:
                y_hat (np.array): A numpy array with the shape (N, )
    """
    assert X.ndim==2
    N = X.shape[0]
    d = X.shape[1]
    X_d_1 = np.insert(X, 0, 1, axis=1)
    y_hat = np.matmul(X_d_1,beta)

    assert beta.shape == (d+1,1)



    y_hat = y_hat.reshape(-1)
    assert y_hat.size == N
    return y_hat
```

[6]:
```
some_X = (np.arange(35).reshape(7,5) ** 13) % 20
some_beta = 2.**(-np.arange(6).reshape(-1,1))
some_yhat = linear_predict(some_X, some_beta)
assert np.array_equal(some_yhat.round(3), np.array([ 3.062,  9.156,  6.188, 15.
 ↪719,  3.062,  9.281,  7.062]))
```

[7]:
```
# Checking against the pre-computed test database
test_results = test_case_checker(linear_predict, task_id=2)
assert test_results['passed'], test_results['message']
```

[8]:
```
# Task 2 Test Cell

# The following are hints to make your life easier duing debugging if you␣
 ↪failed the pre-computed tests.
#
#   When an error is raised in checking against the pre-computed test database:
#
#     0. test_results will be a python dictionary, with the bug information␣
 ↪stored in it. Don't be afraid to look into it!
#
#     1. You can access the failed test arguments by reading␣
 ↪test_results['test_kwargs']. test_results['test_kwargs'] will be
#        another python dictionary with its keys being the argument names and␣
 ↪the values being the argument values.
#
#     2. test_results['correct_sol'] will contain the correct solution.
#
#     3. test_results['stu_sol'] will contain your implementation's returned␣
 ↪solution.
```

4 Task 3

Using the `linear_predict` function that you previously wrote, write a function `linear_residuals` that given the learned weights in the `linear_regression` function calculates the residuals vector. Your functions takes the following arguments as input:

1. `X`: A numpy array of the shape `(N,d)` where `N` is the number of data points, and `d` is the data dimension. Do not assume anything about `N` or `d` other than being a positive integer.
2. `beta`: A numpy array of the shape `(d+1,1)` where `d` is the data dimension

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \cdots \\ \beta_d \end{bmatrix}$$

3. `Y`: A numpy array of the shape `(N,)` where `N` is the number of data points.

Your function should produce the e numpy array with the shape of `(N,)`, whose i^{th} element is defined as

$$e^{(i)} = y^{(i)} - (\beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + \cdots + \beta_d x_d^{(i)})$$

.

[9]:
```
def linear_residuals(X,beta,Y):
    """
    Calculate residual vector using linear_predict function

        Parameters:
```

```
            X (np.array): A numpy array with the shape (N, d) where N is␣
↪the number of data points and d is dimension
            Y (np.array): A numpy array with the shape (N, ) where N is the␣
↪number of data points
            beta (np.array): A numpy array of the shape (d+1,1) where d is␣
↪the data dimension

    Returns:
            e (np.array): A numpy array with the shape (N, ) that␣
↪represents the residual vector
    """

    assert X.ndim==2
    N = X.shape[0]
    d = X.shape[1]
    assert beta.shape == (d+1,1)
    assert Y.shape == (N,)


    e  = Y - linear_predict(X,beta)

    e = e.reshape(-1)
    assert e.size == N
    return e
```

[10]:
```
some_X = (np.arange(35).reshape(7,5) ** 13) % 20
some_beta = 2.**(-np.arange(6).reshape(-1,1))
some_Y = np.sum(some_X, axis=1)
some_res = linear_residuals(some_X, some_beta, some_Y)
assert np.array_equal(some_res.round(3), np.array([16.938, 35.844, 33.812, 59.
↪281, 16.938, 39.719, 16.938]))
```

[11]:
```
# Checking against the pre-computed test database
test_results = test_case_checker(linear_residuals, task_id=3)
assert test_results['passed'], test_results['message']
```

[ ]:
```
# Task 3 Test Cell

# The following are hints to make your life easier duing debugging if you␣
↪failed the pre-computed tests.
#
#   When an error is raised in checking against the pre-computed test database:
#
#     0. test_results will be a python dictionary, with the bug information␣
↪stored in it. Don't be afraid to look into it!
#
```

```
#     1. You can access the failed test arguments by reading␣
 ↪test_results['test_kwargs']. test_results['test_kwargs'] will be
#        another python dictionary with its keys being the argument names and␣
 ↪the values being the argument values.
#
#     2. test_results['correct_sol'] will contain the correct solution.
#
#     3. test_results['stu_sol'] will contain your implementation's returned␣
 ↪solution.
```

5 1. Problem 1

5.1 1.0 Data

5.1.1 1.0.1 Description

A dataset containing the blood sulfate measured in a Baboon can be found at http://www.statsci.org/data/general/brunhild.html. The observations are recorded as a function of time and there are 20 records in the file.

5.1.2 1.0.2 Information Summary

- **Input/Output**: This data has two columns; the first is the time of measurement with the unit being an hour since the radioactive material injection, and the second column is the blood sulfate levels in the unit of Geiger counter counts times 10^{-4}.
- **Missing Data**: There is no missing data.
- **Final Goal**: We want to **properly** fit a linear regression model.

5.1.3 1.0.3 Loading The Data

[12]:
```
df_1 = pd.read_csv('brunhild.txt', sep='\t')
df_1
```

[12]:

	Hours	Sulfate
0	2	15.11
1	4	11.36
2	6	9.77
3	8	9.09
4	10	8.48
5	15	7.69
6	20	7.33
7	25	7.06
8	30	6.70
9	40	6.43
10	50	6.16
11	60	5.99
12	70	5.77

```
13     80     5.64
14     90     5.39
15    110     5.09
16    130     4.87
17    150     4.60
18    160     4.50
19    170     4.36
20    180     4.27
```

5.2 1.1 Regression

We apply linear regression to this dataset. First, in Section 1.1.1. we apply linear regression to the original coordinates, and then in Section 1.1.2. we apply linear regression in the log-log coordinate. You should see the results and compare them. We use the code that you implemented in the previous tasks.

Attention: Although you are not adding any code in this part, you should see the results, compare them, and think about what is going on. Moreover, you might need to come back and modify the code to answer some questions in the follow-up quiz.

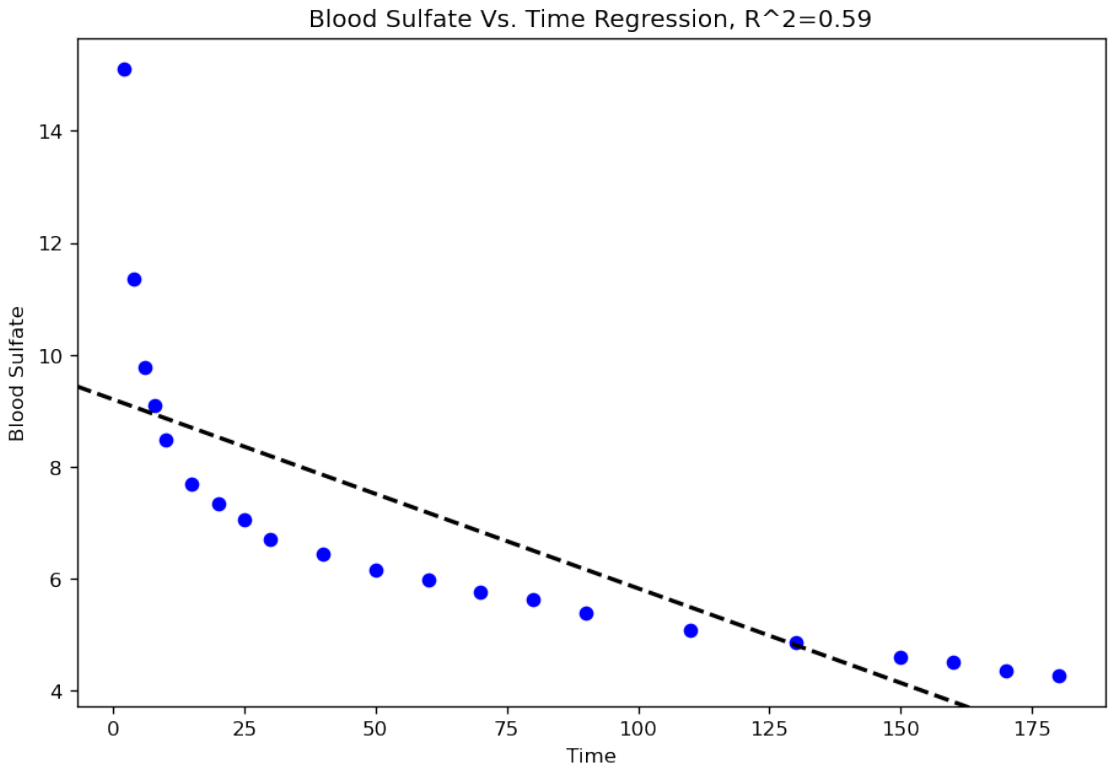
The following two functions will be useful to draw regression plots.

[13]:
```
def newline(p1, p2, ax):
    # This code was borrowed from
    # https://stackoverflow.com/questions/36470343/
 ↪how-to-draw-a-line-with-matplotlib/36479941
    xmin, xmax = ax.get_xbound()

    if(p2[0] == p1[0]):
        xmin = xmax = p1[0]
        ymin, ymax = ax.get_ybound()
    else:
        ymax = p1[1]+(p2[1]-p1[1])/(p2[0]-p1[0])*(xmax-p1[0])
        ymin = p1[1]+(p2[1]-p1[1])/(p2[0]-p1[0])*(xmin-p1[0])

    l = mlines.Line2D([xmin,xmax], [ymin,ymax])
    ax.add_line(l)
    return l

def draw_regression(X,Y,beta,ax):
    ax.scatter(X, Y, c='b', marker='o')
    line_obj = newline([0, np.sum(beta*np.array([[1],[0]]))], [2, np.
 ↪sum(beta*np.array([[1],[2]]))], ax)
    line_obj.set_color('black')
    line_obj.set_linestyle('--')
    line_obj.set_linewidth(2)
    return ax
```

5.2.1 1.1.1 Regression in the Original Coordinates

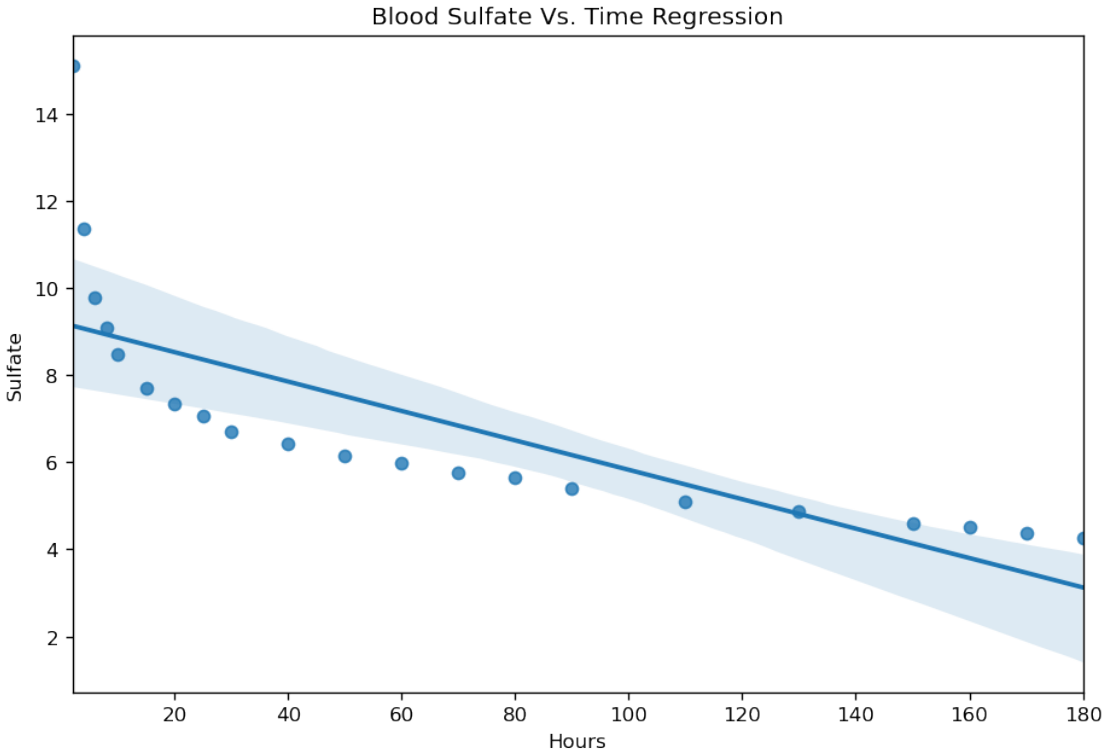
Now, we find the linear regression in the original coordinates. For this, we use the `linear_regression` and `linear_residuals` functions that you implemented previously. We do not use any regularization here, so $\lambda = 0$.

[14]:
```
X_1 = df_1['Hours'].values.reshape(-1,1)
Y_1 = df_1['Sulfate'].values.reshape(-1)

fig, ax = plt.subplots(figsize=(9,6.), dpi=120)

beta_1 = linear_regression(X_1,Y_1,lam=0)
ax = draw_regression(X_1,Y_1,beta_1,ax)

residuals_1 = linear_residuals(X_1, beta_1, Y_1)
fitted_1 = linear_predict(X_1, beta_1)

r2_1 = r2_score(Y_1, fitted_1) #computes the R^2 score

ax.set_xlabel('Time')
ax.set_ylabel('Blood Sulfate')
_ = ax.set_title('Blood Sulfate Vs. Time Regression, R^2=%.2f' %r2_1)
```



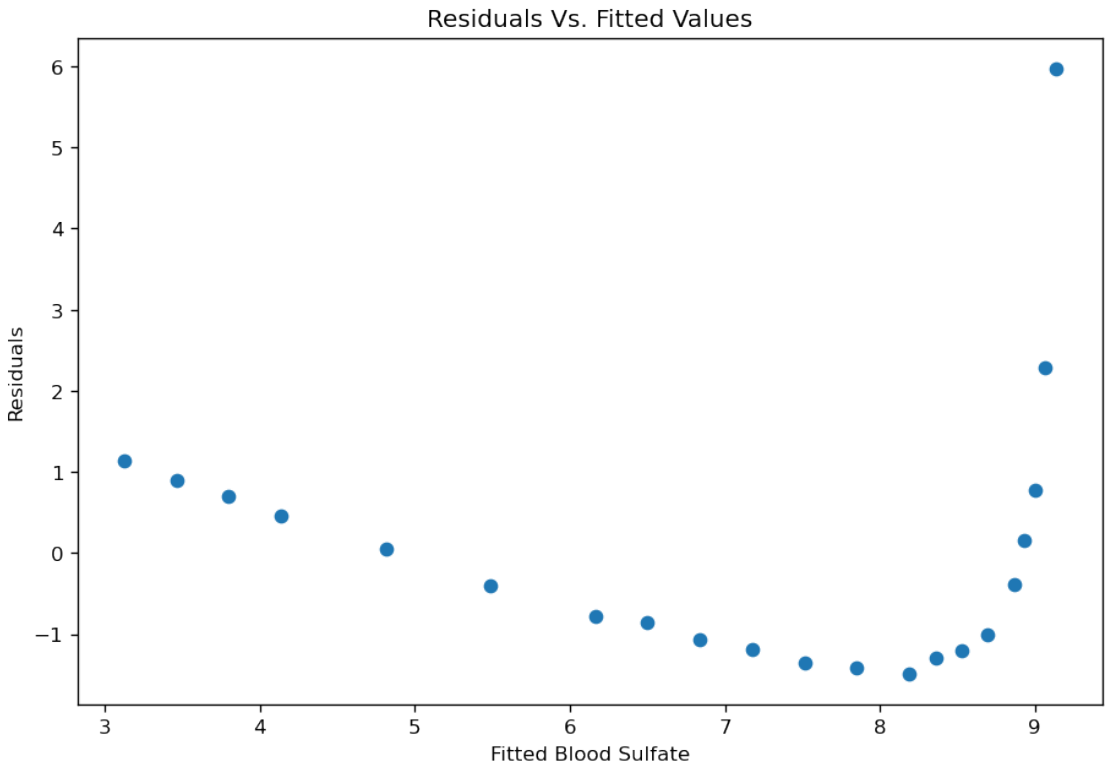
Lets compare our result with an off-the-shelf package. The package**seaborn** does the whole linear regression process in a single line. Let's try that, and see how it matches with our plot.

[15]:
```
fig, ax = plt.subplots(figsize=(9,6.), dpi=120)
sns.regplot(x='Hours', y='Sulfate', data=df_1, ax=ax)
_ = ax.set_title('Blood Sulfate Vs. Time Regression')
```



Now we draw the residuals against the fitted values.

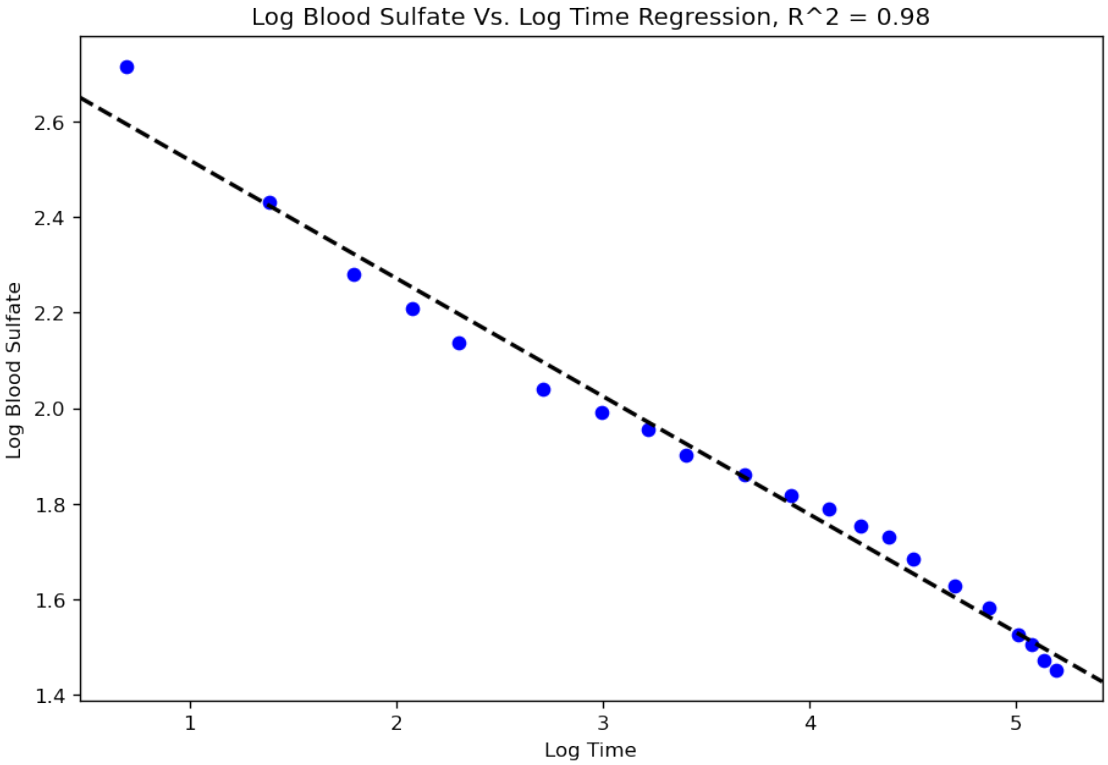
[16]:
```
fig, ax = plt.subplots(figsize=(9,6.), dpi=120)
ax.scatter(fitted_1, residuals_1)

ax.set_xlabel('Fitted Blood Sulfate')
ax.set_ylabel('Residuals')
_ = ax.set_title('Residuals Vs. Fitted Values')
```



5.2.2 1.1.2 Regression in the Log-Log Coordinates

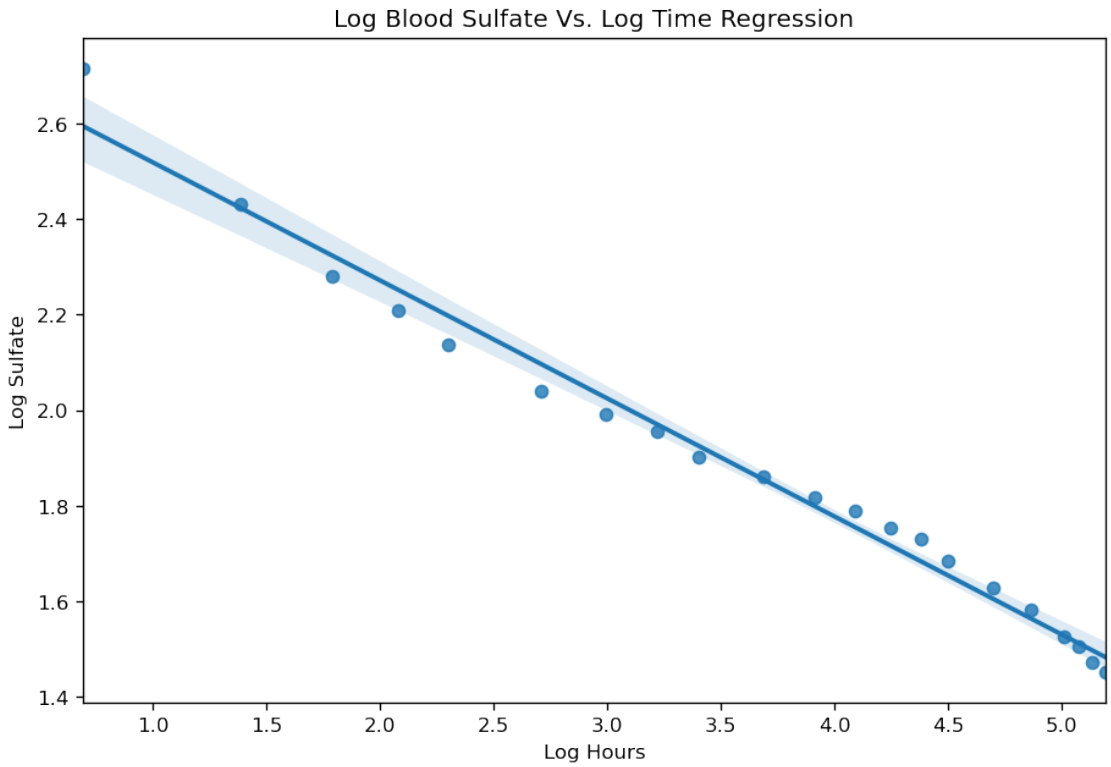
Next, we find the linear regression for the log of the blood sulfate level against the log of time. We first use the `linear_regression` and `linear_residuals` functions that you implemented above.

[17]:
```
log_X_1 = np.log(df_1['Hours'].values.reshape(-1,1))
log_Y_1 = np.log(df_1['Sulfate'].values.reshape(-1))

fig, ax = plt.subplots(figsize=(9,6.), dpi=120)

beta_1_log = linear_regression(log_X_1,log_Y_1,lam=0)
residuals_1_log = linear_residuals(log_X_1, beta_1_log, log_Y_1)
fitted_1_log = linear_predict(log_X_1, beta_1_log)

r2_1_log = r2_score(log_Y_1, fitted_1_log) #computes the R^2 score

ax = draw_regression(log_X_1,log_Y_1,beta_1_log,ax)

ax.set_xlabel('Log Time')
ax.set_ylabel('Log Blood Sulfate')
_ = ax.set_title('Log Blood Sulfate Vs. Log Time Regression, R^2 = %.2f'␣
 ↪%r2_1_log)
```



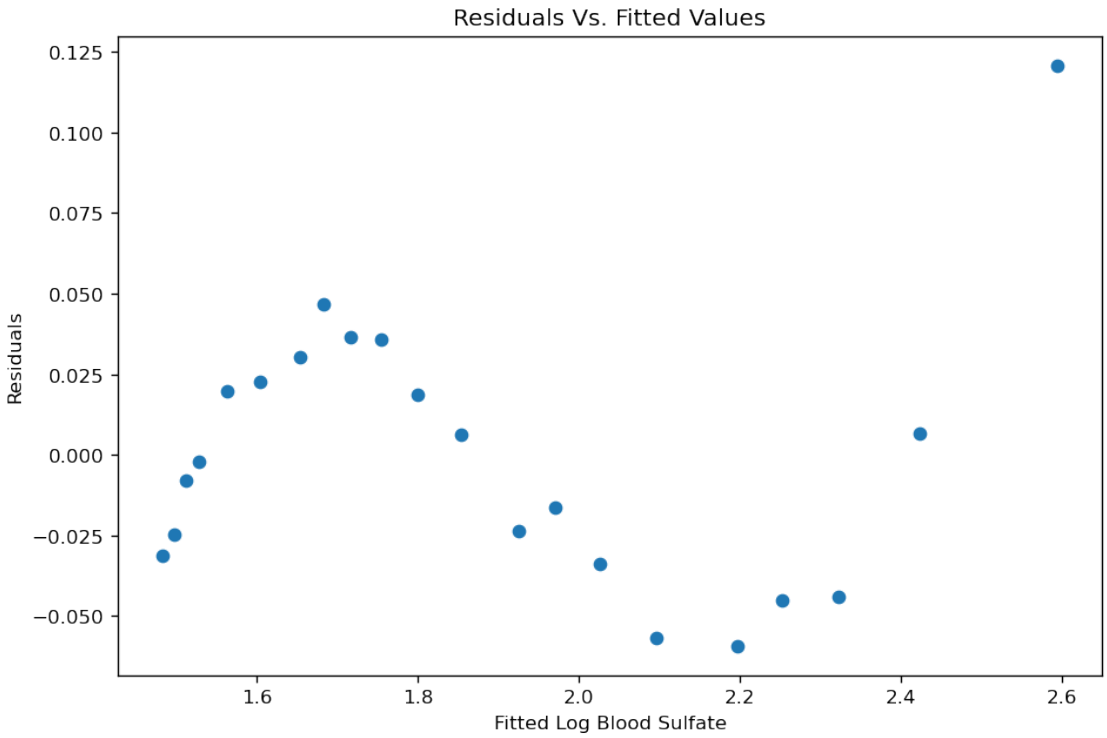
We also compare our plot with the `seaborn` package.

[18]:
```
fig, ax = plt.subplots(figsize=(9,6.), dpi=120)
log_df_1 = df_1.copy(deep=True)
log_df_1['Log Hours'] = np.log(df_1['Hours'])
log_df_1['Log Sulfate'] = np.log(df_1['Sulfate'])
sns.regplot(x='Log Hours', y='Log Sulfate', data=log_df_1, ax=ax)
_ = ax.set_title('Log Blood Sulfate Vs. Log Time Regression')
```



We also plot the residuals against fitted log blood sulfate.

[19]:
```
fig, ax = plt.subplots(figsize=(9,6.), dpi=120)
ax.scatter(fitted_1_log, residuals_1_log)

ax.set_xlabel('Fitted Log Blood Sulfate')
ax.set_ylabel('Residuals')
_ = ax.set_title('Residuals Vs. Fitted Values')
```



6 2. Problem 2

6.1 2.0 Data

6.1.1 2.0.1 Description

At http://www.statsci.org/data/oz/physical.html, you will find a dataset of measurements by M. Larner, made in 1996. These measurements include body mass, and various diameters. Build a linear regression of predicting the body mass from these diameters.

6.1.2 2.0.2 Information Summary

- **Input/Output**: This data has 11 columns, with the first column being the body mass and label.
- **Missing Data**: There is no missing data.
- **Final Goal**: We want to fit a linear regression model.

6.1.3 2.0.3 Loading The Data

[20]:
```
df_2 = pd.read_csv('physical.txt', sep='\t')
df_2
```

[20]:

	Mass	Fore	Bicep	Chest	Neck	Shoulder	Waist	Height	Calf	Thigh	Head
0	77.0	28.5	33.5	100.0	38.5	114.0	85.0	178.0	37.5	53.0	58.0
1	85.5	29.5	36.5	107.0	39.0	119.0	90.5	187.0	40.0	52.0	59.0
2	63.0	25.0	31.0	94.0	36.5	102.0	80.5	175.0	33.0	49.0	57.0
3	80.5	28.5	34.0	104.0	39.0	114.0	91.5	183.0	38.0	50.0	60.0
4	79.5	28.5	36.5	107.0	39.0	114.0	92.0	174.0	40.0	53.0	59.0
5	94.0	30.5	38.0	112.0	39.0	121.0	101.0	180.0	39.5	57.5	59.0
6	66.0	26.5	29.0	93.0	35.0	105.0	76.0	177.5	38.5	50.0	58.5
7	69.0	27.0	31.0	95.0	37.0	108.0	84.0	182.5	36.0	49.0	60.0
8	65.0	26.5	29.0	93.0	35.0	112.0	74.0	178.5	34.0	47.0	55.5
9	58.0	26.5	31.0	96.0	35.0	103.0	76.0	168.5	35.0	46.0	58.0
10	69.5	28.5	37.0	109.5	39.0	118.0	80.0	170.0	38.0	50.0	58.5
11	73.0	27.5	33.0	102.0	38.5	113.0	86.0	180.0	36.0	49.0	59.0
12	74.0	29.5	36.0	101.0	38.5	115.5	82.0	186.5	38.0	49.0	60.0
13	68.0	25.0	30.0	98.5	37.0	108.0	82.0	188.0	37.0	49.5	57.0
14	80.0	29.5	36.0	103.0	40.0	117.0	95.5	173.0	37.0	52.5	58.0
15	66.0	26.5	32.5	89.0	35.0	104.5	81.0	171.0	38.0	48.0	56.5
16	54.5	24.0	30.0	92.5	35.5	102.0	76.0	169.0	32.0	42.0	57.0
17	64.0	25.5	28.5	87.5	35.0	109.0	84.0	181.0	35.5	42.0	58.0
18	84.0	30.0	34.5	99.0	40.5	119.0	88.0	188.0	39.0	50.5	56.0
19	73.0	28.0	34.5	97.0	37.0	104.0	82.0	173.0	38.0	49.0	58.0
20	89.0	29.0	35.5	106.0	39.0	118.0	96.0	179.0	39.5	51.0	58.5
21	94.0	31.0	33.5	106.0	39.0	120.0	99.5	184.0	42.0	55.0	57.0

6.2 2.1 Regression

6.2.1 2.1.1 Original Coordinates

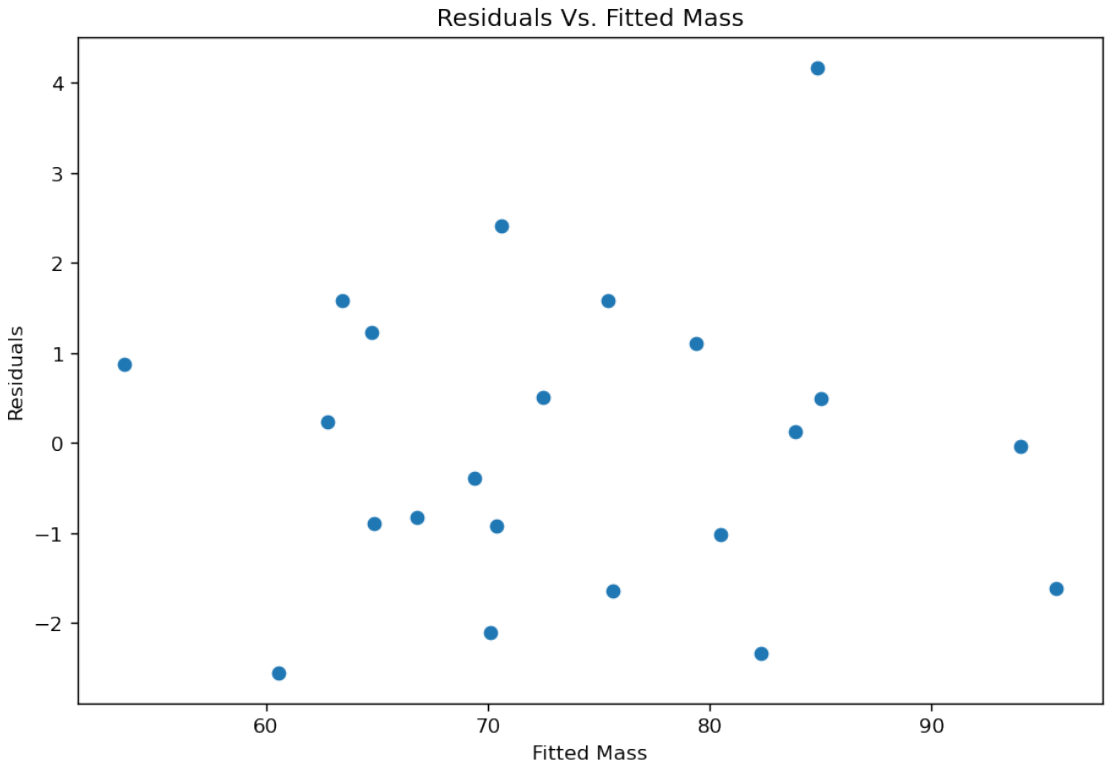
We first try to find the linear regression to predict the body mass based on the input diameters in the original coordinates. Note that unlike Problem 1, we have 11 input variables here, and we cannot plot body mass against the input variables and see how the fitted plot behaves. For this, we plot the residuals against the fitted mass. Similar to Problem 1, we do not use regularization and hence $\lambda = 0$.

Attention: Although you are not adding any code in this part, you should see the results, compare them, and think about what is going on. Moreover, you might need to come back and modify the code to answer some questions in the follow-up quiz.

[21]:
```
X_2 = df_2.loc[:, df_2.columns != 'Mass'].values
Y_2 = df_2['Mass'].values

fig, ax = plt.subplots(figsize=(9,6.), dpi=120)

beta_2 = linear_regression(X_2,Y_2,lam=0)
residuals_2 = linear_residuals(X_2, beta_2, Y_2)
fitted_2 = linear_predict(X_2, beta_2)

ax.scatter(fitted_2, residuals_2)
```

```
ax.set_xlabel('Fitted Mass')
ax.set_ylabel('Residuals')
_ = ax.set_title('Residuals Vs. Fitted Mass')

print('mean square error: %.2f' %np.mean(residuals_2**2))
```

```
mean square error: 2.61
```

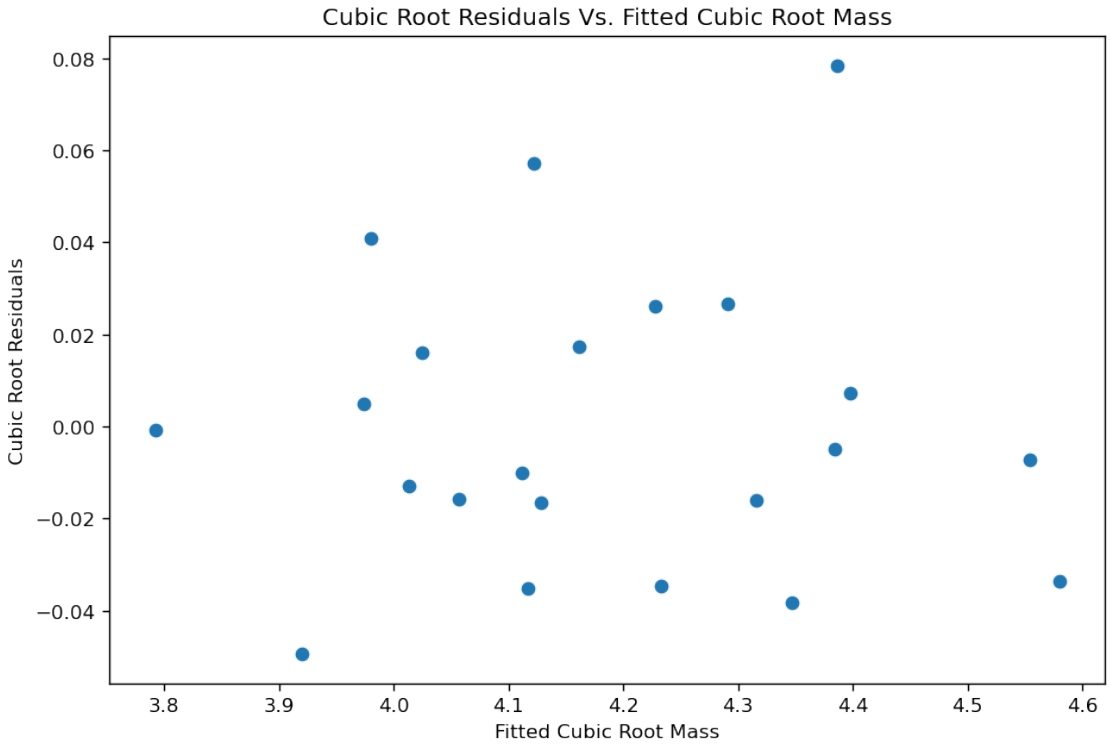


6.2.2 2.1.2 Cubic Root Labels

Now, we find the linear regression between the input variables and the cubic root of the body mass. Then, we plot cubic root residuals against fitted cubic root mass.

[22]:
```
X_2 = df_2.loc[:, df_2.columns != 'Mass'].values
Y_2_cr = (df_2['Mass'].values**(1./3.))

fig, ax = plt.subplots(figsize=(9,6.), dpi=120)

beta_2_cr = linear_regression(X_2,Y_2_cr,lam=0)
residuals_2_cr = linear_residuals(X_2, beta_2_cr, Y_2_cr)
fitted_2_cr = linear_predict(X_2, beta_2_cr)
```

```
ax.scatter(fitted_2_cr, residuals_2_cr)

ax.set_xlabel('Fitted Cubic Root Mass')
ax.set_ylabel('Cubic Root Residuals')
_ = ax.set_title('Cubic Root Residuals Vs. Fitted Cubic Root Mass')
```



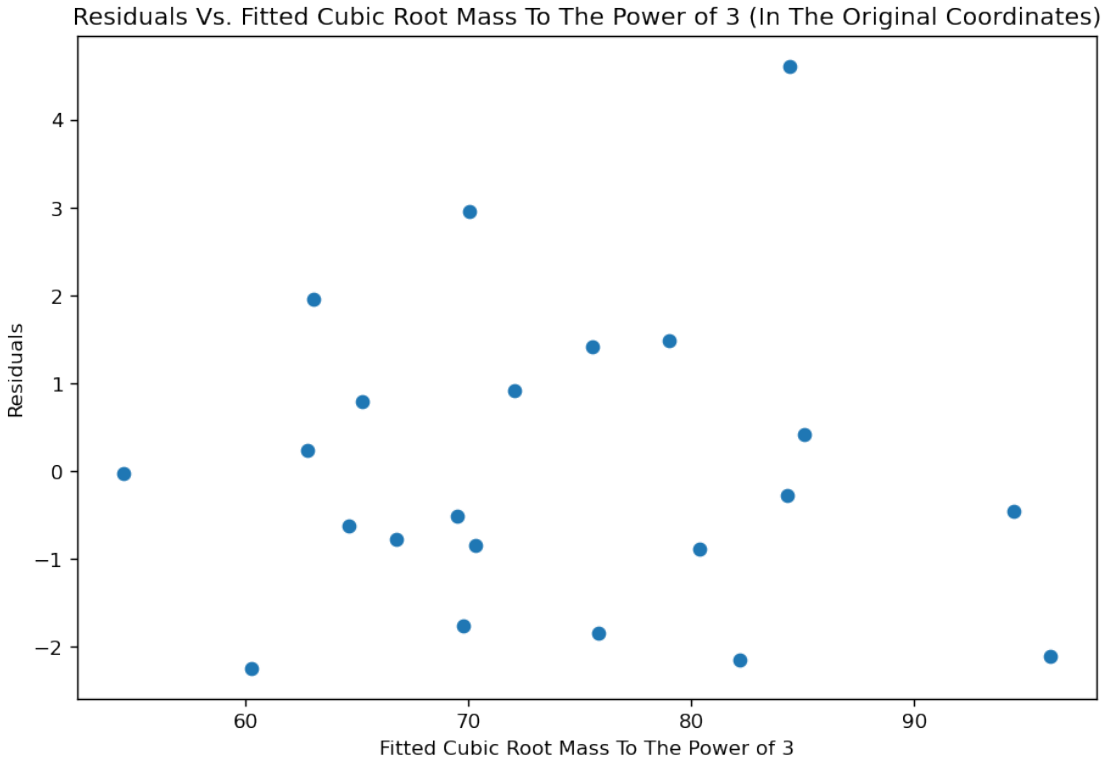
6.2.3 2.1.3 Cubic Root Labels in the Original Scale

To compare the fitted values in the original scale, we raise the fitted cubic root mass to the power of 3 and compare them with the original mass values. Then, we plot the residuals against fitted cubic root mass to the power of 3.

[23]:
```
X_2 = df_2.loc[:, df_2.columns != 'Mass'].values
Y_2 = df_2['Mass'].values
Y_2_cr = (Y_2**(1./3.))

fig, ax = plt.subplots(figsize=(9,6.), dpi=120)

beta_2_cr = linear_regression(X_2, Y_2_cr, lam=0)
fitted_2_orig = (linear_predict(X_2, beta_2_cr))**3.
residuals_2_orig = Y_2 - fitted_2_orig
```

```
ax.scatter(fitted_2_orig, residuals_2_orig)

ax.set_xlabel('Fitted Cubic Root Mass To The Power of 3')
ax.set_ylabel('Residuals')
_ = ax.set_title('Residuals Vs. Fitted Cubic Root Mass To The Power of 3 (In␣
 ↪The Original Coordinates)')

print('mean square error: %.2f' %np.mean(residuals_2_orig**2))
```

```
mean square error: 2.88
```



7 3. Problem 3

7.1 3.0 Data

7.1.1 3.0.1 Description

At https://archive.ics.uci.edu/ml/datasets/Abalone, you will find a dataset of measurements by W. J. Nash, T. L. Sellers, S. R. Talbot, A. J. Cawthorn and W. B. Ford, made in 1992. These are a variety of measurements of blacklip abalone (Haliotis rubra; delicious by repute) of various ages and genders.

7.1.2 3.0.2 Information Summary

- **Input/Output**: This data has 9 columns, with the last column being the rings count which serves as the age of the abalone and the label.
- **Missing Data**: There is no missing data.
- **Final Goal**: We want to fit a linear regression model predicting the age.

7.1.3 3.0.3 Loading The Data

[24]:
```
df_3 = pd.read_csv('abalone.data', sep=',', header=None)
df_3.columns = ['Sex', 'Length', 'Diameter', 'Height', 'Whole weight', 'Shucked␣
↪weight',
                'Viscera weight', 'Shell weight', 'Rings']
df_3
```

[24]:

	Sex	Length	Diameter	Height	Whole weight	Shucked weight	\
0	M	0.455	0.365	0.095	0.5140	0.2245	
1	M	0.350	0.265	0.090	0.2255	0.0995	
2	F	0.530	0.420	0.135	0.6770	0.2565	
3	M	0.440	0.365	0.125	0.5160	0.2155	
4	I	0.330	0.255	0.080	0.2050	0.0895	
…	..	…	…	…	…	…	
4172	F	0.565	0.450	0.165	0.8870	0.3700	
4173	M	0.590	0.440	0.135	0.9660	0.4390	
4174	M	0.600	0.475	0.205	1.1760	0.5255	
4175	F	0.625	0.485	0.150	1.0945	0.5310	
4176	M	0.710	0.555	0.195	1.9485	0.9455	

	Viscera weight	Shell weight	Rings
0	0.1010	0.1500	15
1	0.0485	0.0700	7
2	0.1415	0.2100	9
3	0.1140	0.1550	10
4	0.0395	0.0550	7
…	…	…	…
4172	0.2390	0.2490	11
4173	0.2145	0.2605	10
4174	0.2875	0.3080	9
4175	0.2610	0.2960	10
4176	0.3765	0.4950	12

[4177 rows x 9 columns]

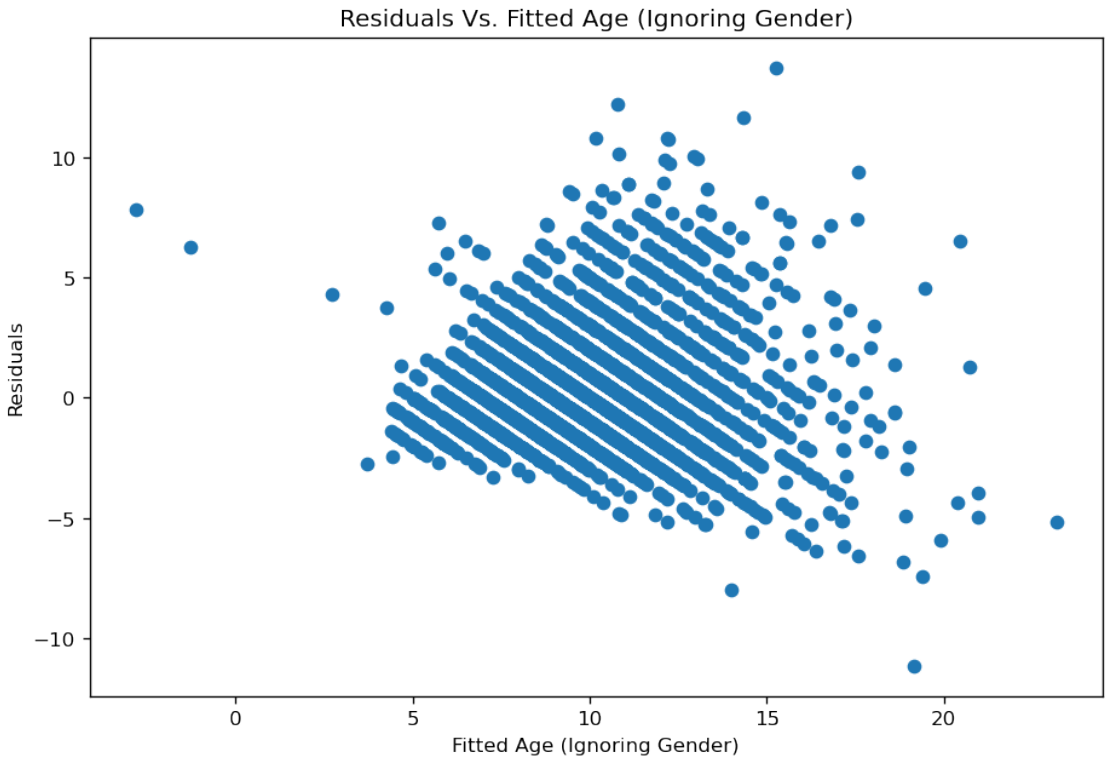
7.2 3.1 Predicting the age from the measurements, ignoring gender

Our goal is to predict the number of rings against the input variables. However, since the input gender variable is discrete (it is one of the three values `M`, `F`, or `I`), we first ignore the gender input.

Attention: Although you are not adding any code in this part, you should see the results, compare them, and think about what is going on. Moreover, you might need to come back and modify the code to answer some questions in the follow-up quiz.

[25]:
```
X_3 = df_3.loc[:, (df_3.columns != 'Rings') & (df_3.columns != 'Sex')].values
Y_3 = df_3['Rings'].values

fig, ax = plt.subplots(figsize=(9,6.), dpi=120)

beta_3 = linear_regression(X_3,Y_3,lam=0)
residuals_3 = linear_residuals(X_3, beta_3, Y_3)
fitted_3 = linear_predict(X_3, beta_3)

ax.scatter(fitted_3, residuals_3)

ax.set_xlabel('Fitted Age (Ignoring Gender)')
ax.set_ylabel('Residuals')
_ = ax.set_title('Residuals Vs. Fitted Age (Ignoring Gender)')

print('mean square error = %.2f' %np.mean(residuals_3**2))
```

```
mean square error = 4.91
```

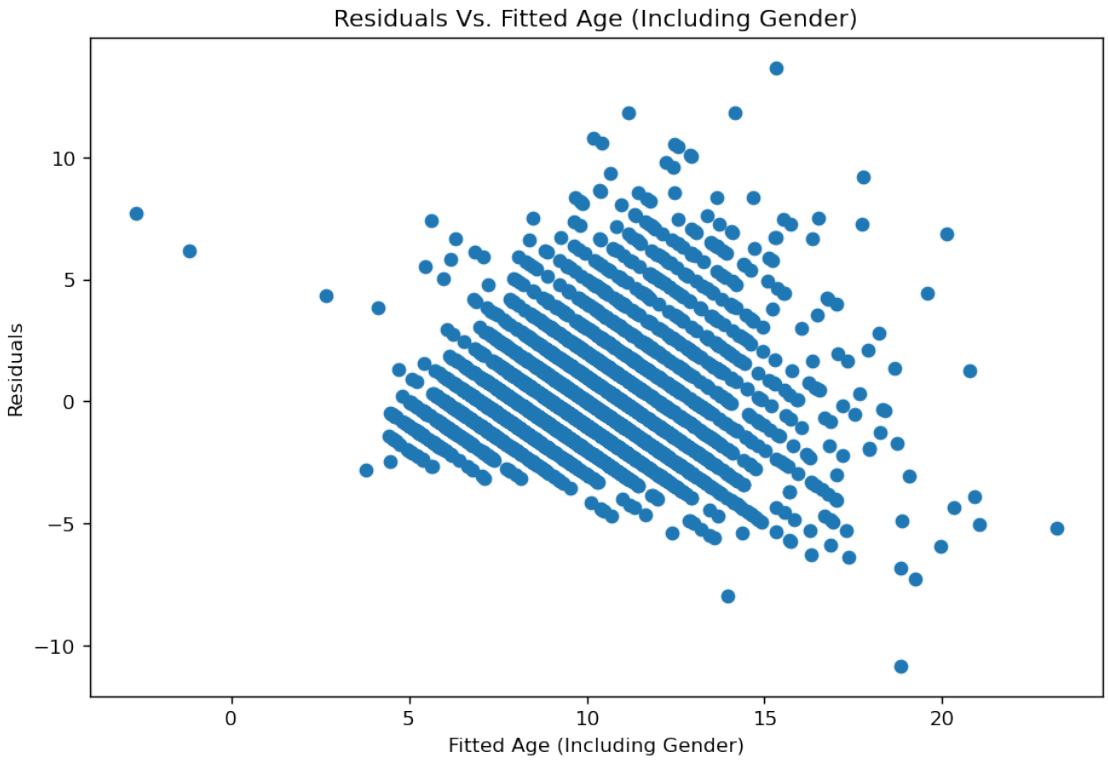


7.3 3.2 Predicting the age from the measurements, including gender

Now, we convert gender into a numeric value by replacing `F` with 1, `M` with 0, and `I` with -1. Then, we again run the linear regression.

[26]:
```
X_3_gender = df_3.loc[:, (df_3.columns != 'Rings') & (df_3.columns != 'Sex')].
 ↪values
X_3_gender = np.concatenate([X_3, np.array([{'F':1, 'M':0, 'I':-1}.get(x) for x␣
 ↪in df_3.Sex]).reshape(-1,1)], axis=1)
Y_3 = df_3['Rings'].values

fig, ax = plt.subplots(figsize=(9,6.), dpi=120)

beta_3_gender = linear_regression(X_3_gender,Y_3,lam=0)
residuals_3_gender = linear_residuals(X_3_gender, beta_3_gender, Y_3)
fitted_3_gender = linear_predict(X_3_gender, beta_3_gender)

ax.scatter(fitted_3_gender, residuals_3_gender)

ax.set_xlabel('Fitted Age (Including Gender)')
ax.set_ylabel('Residuals')
_ = ax.set_title('Residuals Vs. Fitted Age (Including Gender)')

print('mean square error = %.2f' %np.mean(residuals_3_gender**2))
```

```
mean square error = 4.85
```

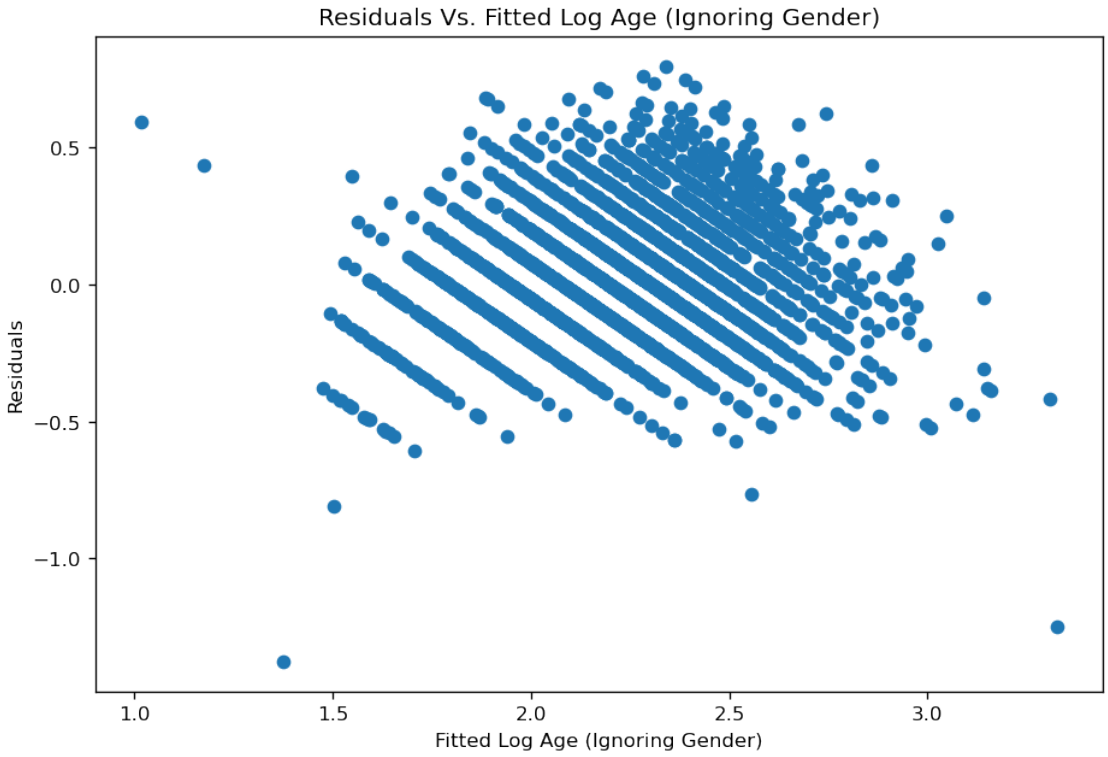


7.4 3.3 Predicting the log of age from the measurements, ignoring gender

We now find the linear regression of the log of the output against the input variables, ignoring gender.

[27]:
```
X_3 = df_3.loc[:, (df_3.columns != 'Rings') & (df_3.columns != 'Sex')].values
Y_3 = df_3['Rings'].values
Y_3_log = np.log(df_3['Rings'].values)

fig, ax = plt.subplots(figsize=(9,6.), dpi=120)

beta_3_log = linear_regression(X_3,Y_3_log,lam=0)
residuals_3_log = linear_residuals(X_3, beta_3_log, Y_3_log)
fitted_3_log = linear_predict(X_3, beta_3_log)

ax.scatter(fitted_3_log, residuals_3_log)

ax.set_xlabel('Fitted Log Age (Ignoring Gender)')
ax.set_ylabel('Residuals')
_ = ax.set_title('Residuals Vs. Fitted Log Age (Ignoring Gender)')

fitted_3_log_orig = np.exp(fitted_3_log) #predicted values back to the original␣
↪coordinates
residuals_3_log_orig = Y_3 - fitted_3_log_orig #residuals in the original␣
↪coordinates
print('mean square error (in the original coordinates) = %.2f' %np.
↪mean(residuals_3_log_orig**2))
```

```
mean square error (in the original coordinates) = 5.31
```



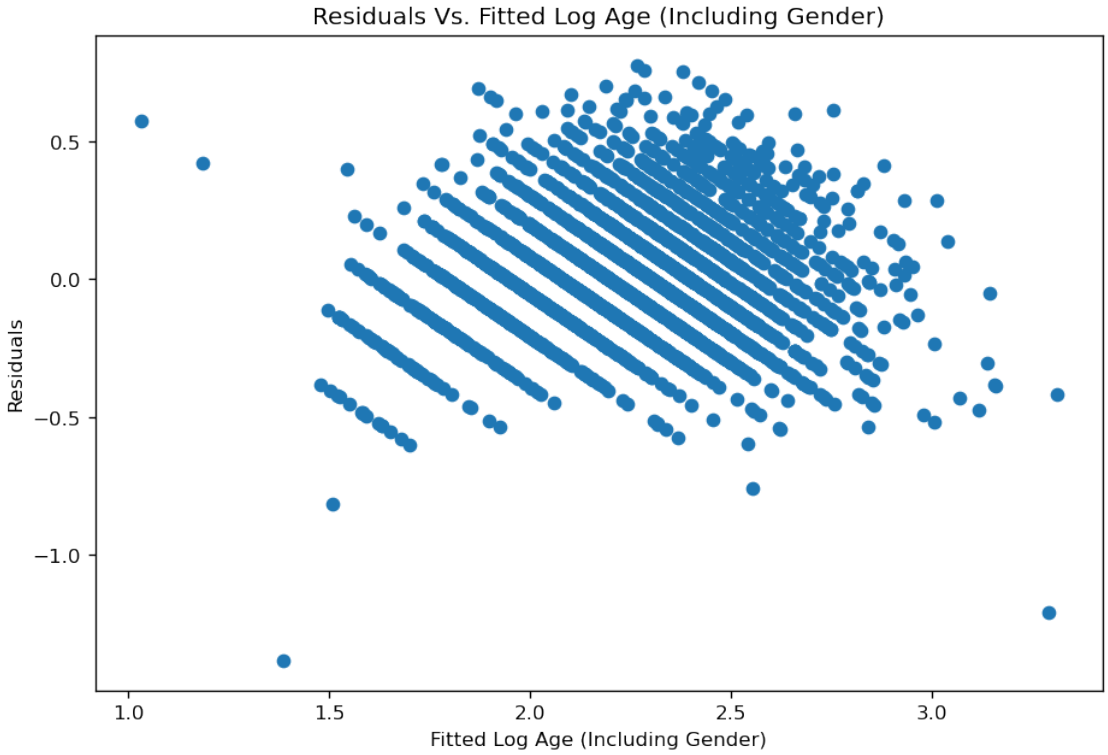
7.5 3.4 Predicting the log age from the measurements, including gender

We use the same numeric values for the gender as in Section 3.2, and find the linear regression to predict the log of the output against the input variables.

[28]:
```
X_3_gender = df_3.loc[:, (df_3.columns != 'Rings') & (df_3.columns != 'Sex')].
 ↪values
X_3_gender = np.concatenate([X_3_gender, np.array([{'F':1, 'M':0, 'I':-1}.
 ↪get(x) for x in df_3.Sex]).reshape(-1,1)], axis=1)
Y_3 = df_3['Rings'].values
Y_3_log = np.log(df_3['Rings'].values)

fig, ax = plt.subplots(figsize=(9,6.), dpi=120)

beta_3_gender_log = linear_regression(X_3_gender,Y_3_log,lam=0)
residuals_3_gender_log = linear_residuals(X_3_gender, beta_3_gender_log,␣
 ↪Y_3_log)
fitted_3_gender_log = linear_predict(X_3_gender, beta_3_gender_log)

ax.scatter(fitted_3_gender_log, residuals_3_gender_log)

ax.set_xlabel('Fitted Log Age (Including Gender)')
ax.set_ylabel('Residuals')
```

```
_ = ax.set_title('Residuals Vs. Fitted Log Age (Including Gender)')

fitted_3_gender_log_orig = np.exp(fitted_3_gender_log) # predicted values back␣
 ↪to the original coordinates
residuals_3_gender_log = Y_3 - fitted_3_gender_log_orig
print('mean square error (in the original coordinates) = %.2f' %np.
 ↪mean(residuals_3_gender_log**2))
```

```
mean square error (in the original coordinates) = 5.24
```



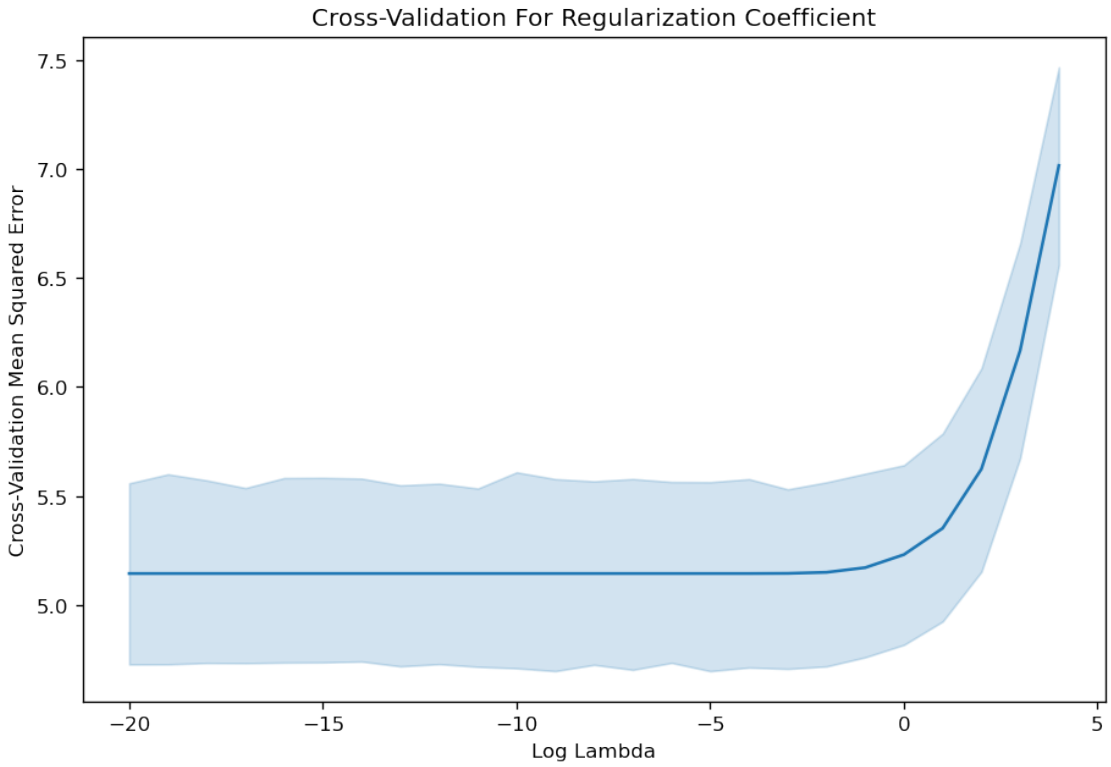
7.6 3.5 Applying Cross Validation For Regularization

We now bring the regularization into play. We use cross validation to find the value of λ to predict the log of the output variable against all the input variables. We convert the gender input to a numeric value using the conversion in Section 3.2. We then plot the cross-validation mean square error against the log of λ.

In the following code, the variable `Y_transform` determines whether we try to predict the labels in the original coordinates or in the logarithmic space. If the value of `Y_transform` is `linear`, we apply linear transformation in the original coordinates, and if its value is `log`, we use logarithmic transformation. In case we work in the logarithmic space, in order to find the mean square error, after finding the predicted values, we transform them back into the original coordinates using `np.exp` and then compare them to the correct values.

[29]:
```
log_lambda_list = np.arange(-20, 5)
X_3_gender = df_3.loc[:, (df_3.columns != 'Rings') & (df_3.columns != 'Sex')].
 ↪values
X_3_gender = np.concatenate([X_3_gender, np.array([{'F':1, 'M':0, 'I':-1}.
 ↪get(x) for x in df_3.Sex]).reshape(-1,1)], axis=1)
Y_3 = df_3['Rings'].values
Y_3_log = np.log(df_3['Rings'].values)

Y_transform = 'log' # 'linear': output variable in the original coordinate is␣
 ↪considered, 'log': log of output is considered

if Y_transform == 'linear':
    X_train_val, X_test, Y_train_val, Y_test = train_test_split(X_3_gender,␣
 ↪Y_3, test_size=0.2, random_state=12345, shuffle=True)
if Y_transform == 'log':
    X_train_val, X_test, Y_train_val, Y_test = train_test_split(X_3_gender,␣
 ↪Y_3_log, test_size=0.2, random_state=12345, shuffle=True)

kf = KFold(n_splits=10, shuffle=True, random_state=12345)

cross_val_mses = []
for train_idx, val_idx in kf.split(X_train_val):
    X_train, X_val, Y_train, Y_val = X_train_val[train_idx],␣
 ↪X_train_val[val_idx], Y_train_val[train_idx], Y_train_val[val_idx]

    for log_lambda in log_lambda_list:
        beta_3_cv = linear_regression(X_train, Y_train, lam=np.exp(log_lambda))
        if Y_transform == 'linear':
            val_residuals = linear_residuals(X_val, beta_3_cv, Y_val)

        if Y_transform == 'log':
            val_predict_log = linear_predict(X_val, beta_3_cv) # predicted in␣
 ↪log space
            val_predict_log_orig = np.exp(val_predict_log) # get back to␣
 ↪original coordinates

            val_residuals = np.exp(Y_val) - val_predict_log_orig

        val_mse = np.mean(val_residuals**2)

        cross_val_mses.append([log_lambda, val_mse])

cross_val_mses = np.array(cross_val_mses)
fig, ax = plt.subplots(figsize=(9,6.), dpi=120)
x_name, y_name = 'Log Lambda', 'Cross-Validation Mean Squared Error'
cv_df = pd.DataFrame(cross_val_mses, columns =[x_name, y_name])
```

```
sns.lineplot(x=x_name, y=y_name,data=cv_df, ax=ax)
_ = ax.set_title('Cross-Validation For Regularization Coefficient')
```



Lets see what was the best value of λ and the corresponding mean square error.

[30]:
```
avg_cv_err_df = cv_df.groupby(x_name).mean()
best_log_lam = avg_cv_err_df[y_name].idxmin()
best_cv_mse = avg_cv_err_df.loc[best_log_lam][y_name]

print(f'Best Log Lambda value was {best_log_lam} with a cross-validation MSE of␣
 ↪%.3f' %best_cv_mse)

beta_full = linear_regression(X_train_val,Y_train_val,lam=np.exp(best_log_lam))

if Y_transform == 'linear':
    test_residuals = linear_residuals(X_test, beta_full, Y_test)
    test_mse = np.mean(test_residuals**2)
if Y_transform == 'log':
    test_predict = linear_predict(X_test, beta_full)
    test_predict_orig = np.exp(test_predict)
    test_residuals = np.exp(Y_test) - test_predict_orig
    test_mse = np.mean(test_residuals**2)
```

```
print(f'The resulting test mean squared error would be %.3f' % test_mse)
```

```
Best Log Lambda value was -5.0 with a cross-validation MSE of 5.144
The resulting test mean squared error would be 10.730
```

[ ]: