DATA 609 - Homework 5: Disciplined Convex Programming and Data Fitting

Eddie Xu

Instructions

Please submit a .qmd file along with a rendered pdf to the Brightspace page for this assignment. You may use whatever language you like within your qmd file, I recommend python, julia, or R.

Problem 1: Penalty Function Approximations (Modified from Exercise 4 in CVX Book Extended Exercises)

Consider the approximation problem:

$$\min_{\mathbf{x}\in\mathbb{R}^n} \phi\left(A\mathbf{x}-\mathbf{b}\right),$$

where A is an $m \times n$ matrix, $x \in \mathbb{R}^n$, and $\phi : \mathbb{R}^m \to \mathbb{R}$ is a convex penalty function measuring the approximation error, and $\mathbf{b}$ is an m-vector.

The purpose of this exercise is for you to implement several different penalty functions in `CVX` and study how the resulting coefficients x from each penalty function differ, as a means of building intuition about penalty functions.

You will use the following penalty functions:

(a) $\phi(\mathbf{y}) = \|y\|_2$, the standard Euclidean norm
(b) $\phi(\mathbf{y}) = \|y\|_1$, the L_1 norm. This is often referred to as the Lasso
(c) $\phi(\mathbf{y}) = \sum_{k=1}^{m/2} |y_{r_k}|$, where r_k is the index of the component with the kth largest absolute value. This is like the Lasso, but where we only count the terms with their error in the top half, i.e. y_{r_1} is the y with largest absolute value, y_{r_2} is the y with second largest absolute value, etc.
(d) $\phi(\mathbf{y}) = \sum_{k=1}^{m} h(y_k)$, where $h(y)$ is the Huber penalty, defined by:

$$h(u) = \begin{cases} u^2, \ |u| \le M \\ M(2|u| - M), \ |u| \ge M, \end{cases}$$

For this problem use $M = 0.2$

(e) $\phi(\mathbf{y}) = \sum_{k=1}^{m} h(y_k)$, where h is the log-barrier penalty, defined by:

$$h(u) = -\log(1 - u^2), \quad \mathbf{dom}(h) = \{u| \quad |u| < 1\}$$

Generate data A and $\mathbf{b}$ as follows:

- $m = 200$
- $n = 100$
- $A_{ij} \sim$ Normal$(\mu = 0, \sigma = 1)$, each element normally distributed with mean 0 and standard deviation 1
- Intialize b as using a normal distribution of mean $\mu = 0$ and $\sigma = 1$, and then normalize b so that all of its entries have absolute value less than 1 by doing something like:
  - $b_i \sim Normal(\mu = 0, \sigma = 1)$
  - and then: `b=b/(1.01 max(abs(b)))`

This is to make sure the `log-barrier` function as a non-empty domain.

Visualize the distribution of errors (using a tool like a histogram or density plot) for each of these penalty function formulations and comment on the differences that you observe. Each penalty function prioritizes a errors differently, how do these priorities manifest in the distribution of residuals.

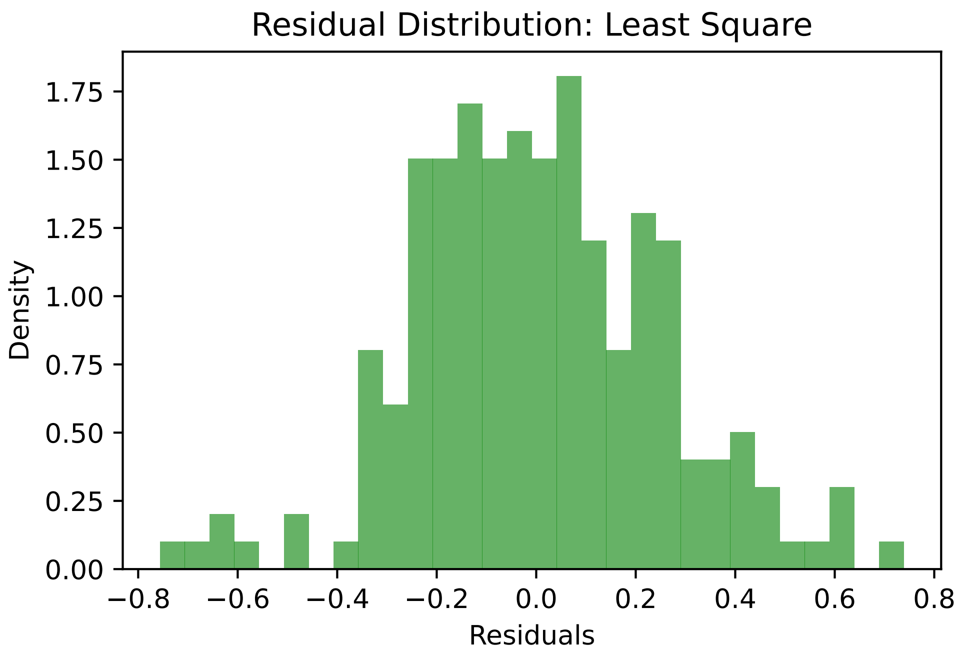
Some hints for selected parts:

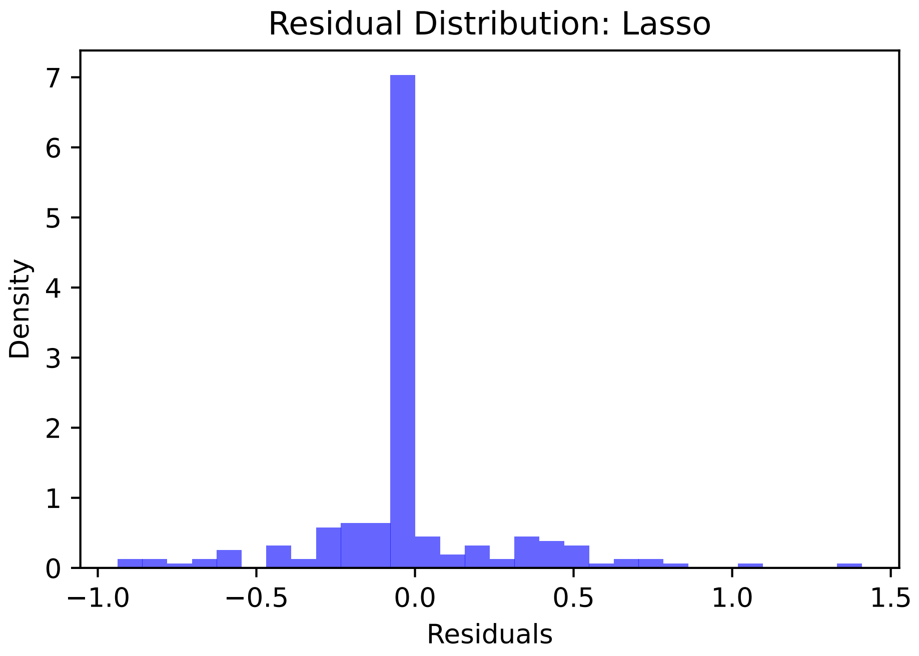
(a) Technically this is a least squares problem, you can solve it using Least-Sqares formula or `CVX`
(b) Use `norm(y,1)`
(c) Use `norm_largest()`
(d) Use `huber()`
(e) The extended exercises claimed that the `log-barrier` objective needed to be reformulated to use the geometric mean, but I found that this problem worked perfectly well with a straightforward implementation. I suspect that the `CVX` software was upgraded to better handle `log` and `exp` objecties since this exercise was developed.

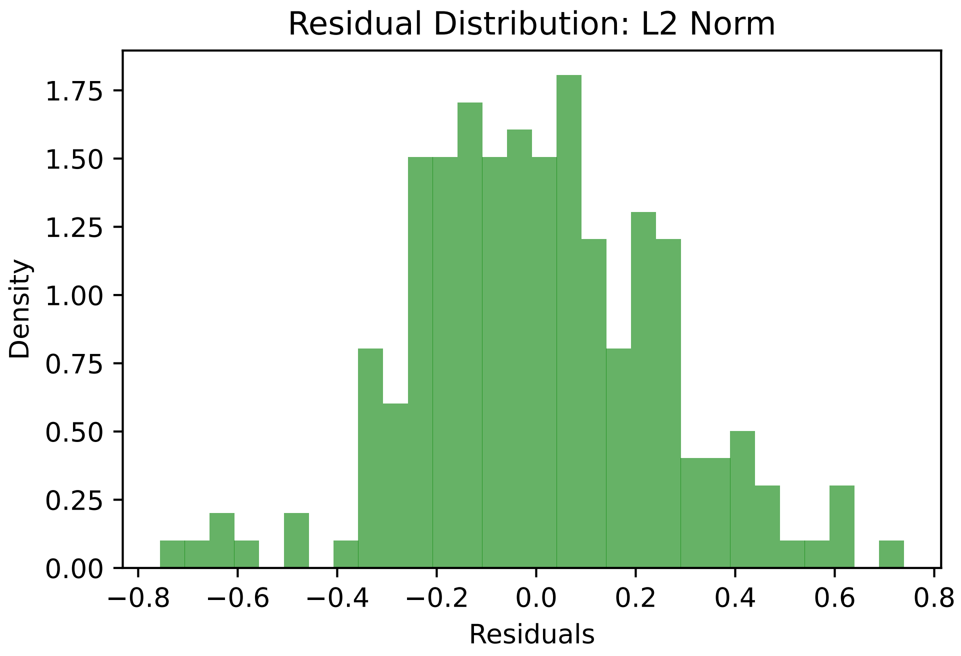
Problem 1 Solution

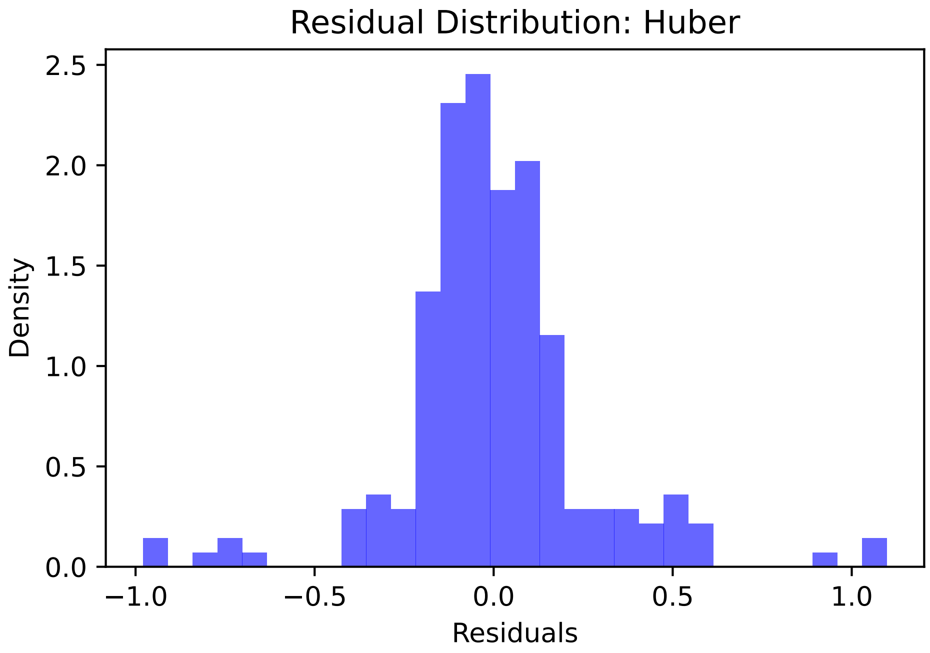
```
# load dependencies
import numpy as np
import pandas as pd
import cvxpy as cp
import matplotlib.pyplot as plt

# define variable and create data
m = 200
n = 100

# create matrix A and normalize b
A = np.random.normal(0,1,(m,n))
b = np.random.normal(0,1,m)
b = b / (1.01 * np.max(np.abs(b)))

# define the optimization variable
x = cp.Variable(n)
y = A @ x - b

# define the penalty function for following:

# (a) Least Square
ls_objective = cp.Minimize(cp.norm(y, 2))
ls_problem = cp.Problem(ls_objective)
ls_problem.solve()
least_square_residual = y.value

# plot (a)
plt.hist(least_square_residual, bins=30, density=True, alpha=0.6, color='g')
plt.title(f'Residual Distribution: Least Square')
plt.xlabel('Residuals')
plt.ylabel('Density')
plt.show()

# (b) L1 Norm
lasso_objective = cp.Minimize(cp.norm(y, 1))
lasso_problem = cp.Problem(lasso_objective)
lasso_problem.solve()
lasso_residual = y.value

# plot (b)
plt.hist(lasso_residual, bins=30, density=True, alpha=0.6, color='b')
```

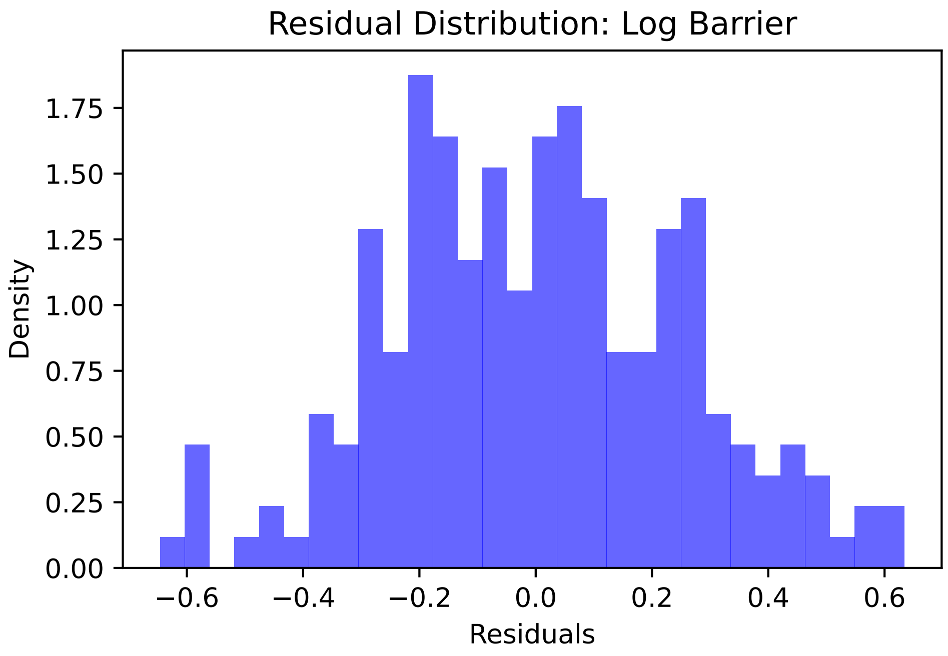
```python
plt.title(f'Residual Distribution: Lasso')
plt.xlabel('Residuals')
plt.ylabel('Density')
plt.show()

# (c) L2 Norm
top_half_objective = cp.Minimize(cp.norm(y, p = 2))
top_half_problem = cp.Problem(top_half_objective)
top_half_problem.solve()
top_half_residual = y.value

# plot (c)
plt.hist(top_half_residual, bins=30, density=True, alpha=0.6, color='g')
plt.title(f'Residual Distribution: L2 Norm')
plt.xlabel('Residuals')
plt.ylabel('Density')
plt.show()

# (d) Huber
M = 0.2
huber_objective = cp.Minimize(sum(cp.huber(y, M)))
huber_problem = cp.Problem(huber_objective)
huber_problem.solve()
huber_residual = y.value

# plot (c)
plt.hist(huber_residual, bins=30, density=True, alpha=0.6, color='b')
plt.title(f'Residual Distribution: Huber')
plt.xlabel('Residuals')
plt.ylabel('Density')
plt.show()

# (e) Log Barrier
log_objective = cp.Minimize(cp.sum([-cp.log(1 - cp.square(r)) for r in y]))
log_problem = cp.Problem(log_objective)
log_problem.solve()
log_residual = y.value

# plot (c)
plt.hist(log_residual, bins=30, density=True, alpha=0.6, color='b')
plt.title(f'Residual Distribution: Log Barrier')
plt.xlabel('Residuals')
plt.ylabel('Density')
plt.show()
```






Residual Distribution: L2 Norm
Density
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
Residuals


Residual Distribution: Huber
Density
2.5
2.0
1.5
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0
Residuals



Problem 2: Fitting Censored Data (Extended Exercises 6.13 in CVX Book)

In some experiments there are two kinds of measurements or data available: The usual ones, in which you get a number (say), and censored data, in which you don't get the specific number, but are told something about it, such as a lower bound.

A classic example is a study of lifetimes of a set of subjects (say, laboratory mice, devices undergoing reliability testing, or people in a long-term, longitudinal study).

For those who have died by the end of data collection, we get the lifetime.

For those who have not died by the end of data collection, we do not have the lifetime, but we do have a lower bound, i.e., the length of the study. In statistics, we call this type of data `right-censored` data, meaning that we do not have the exact values in the right tail of the distribution. The data points that are not present are called the censored data values.

We wish to fit a set of data points, $((\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_k, y_k))$, with $\mathbf{x}_k \in \mathbb{R}^n$ and $y_k \in \mathbb{R}$, with a linear model of the form $y \approx \mathbf{c}^T\mathbf{x}$. The vector $\mathbf{c} \in \mathbf{R}^n$ is the model parameter, which we want to choose. We will use a least-squares criterion, i.e., choose $\mathbf{c}$ to minimize:

$$J = \sum_{i=1}^{k} \left(y_i - \mathbf{c}^T\mathbf{x}_i\right)^2$$

Here is the tricky part: some of the values of y_i are censored; for these entries, we have only a (given) lower bound.

We will re-order the data so that $y_1, \cdots, y_m$ are given (i.e., uncensored), while $y_{m+1}, \cdots y_k$ are all censored, i.e., unknown, but larger than D, a given number. All the values of $\mathbf{x}_i$ are known.

(a) Explain how to find **c** (the model parameter) and $y_{m+1}, \cdots, y_k$ (the censored data values) that minimize J. Hint: should the censored data be variables or parameters?

(b) Carry out the method of part (a) on the data values in the file censored_dict.json. You can process this file in `R` using `fromJSON` in the `jsonlite` package or in python using the `json` library.

Report $\hat{\mathbf{c}}$, the value of **c** found using this method.

Also find $\hat{\mathbf{c}_{ls}}$, the least-squares estimate of **c** obtained by simply ignoring the censored data samples, i.e., the least-squares estimate based on the data $(\mathbf{x}_1, y_1), \cdots (\mathbf{x}_m, y_m)$.

The data file contains $\mathbf{c}_{\text{true}}$, the true value of **c**, in the vector $\mathbf{c}_{\text{true}}$. Use this to give the two relative errors:

$$\frac{\|\mathbf{c}_{\text{true}} - \hat{\mathbf{c}}\|_2^2}{\|\mathbf{c}_{\text{true}}\|_2^2}, \quad \frac{\|\mathbf{c}_{\text{true}} - \hat{\mathbf{c}_{ls}}\|_2^2}{\|\mathbf{c}_{\text{true}}\|_2^2}$$

Problem 2 Solution

Censored data points should not be treated as variables because their exact values are unknown. They should be treated as parameters within the optimization model that satisfy constraints. For censor data points, they will be minimized with the direct error and for censored data points, the constraint will be set as

```
# load dependencies
import numpy as np
import cvxpy as cp
import json
import requests

# load data
url = 'https://raw.githubusercontent.com/georgehagstrom/DATA609Spring2025/refs/heads/main/we
resp = requests.get(url)
data = json.loads(resp.text)

# define variable from data
X = np.array(data["X"]).reshape(data["K"], data["n"])
```

```
y = np.array(data["y"]).flatten()
D = data["D"][0]
c_true = np.array(data["c_true"]).flatten()
n = data["n"]
m = data["M"]
k = data["K"]

# split the data into uncensored and censored data points
X_uncensored = X[:m, :]
y_uncensored = y[:m]
X_censored = X[m:, :]

# set the variable for censored data point and parameter
y_censored = cp.Variable(k - m)
c = cp.Variable(n)

# define the objective for the least-squares error for uncensored data and the error for cens
objective = cp.Minimize(cp.sum_squares(y_uncensored - X_uncensored @ c) + cp.sum_squares(y_ce

# define the constraint
constraints = [y_censored >= D]

# create and solve the optimization problem
problem = cp.Problem(objective, constraints)
problem.solve()

# save the result
c_hat = c.value
y_hat_censored = y_censored.value

# calculate the least-squares estimate ignoring censored data
c_ls = np.linalg.lstsq(X_uncensored, y_uncensored, rcond=None)[0]

# calculate the relative errors
error_c = np.linalg.norm(c_true - c_hat)**2 / np.linalg.norm(c_true)**2
error_c_ls = np.linalg.norm(c_true - c_ls)**2 / np.linalg.norm(c_true)**2

# print
print(f"Optimized c (with censored data): {c_hat}")
print(f"Least-squares c (ignoring censored data): {c_ls}")
print(f"Relative error (with censored data): {error_c}")
print(f"Relative error (least-squares): {error_c_ls}")
```

```
Optimized c (with censored data): [ 0.6595104  -0.9155868  -0.42263851  0.07597687  0.685533
 -0.25297389 -0.48178645  1.81935383  1.62821018 -1.78368885  0.36148809
 -0.58994331  0.18237558  0.72621056 -0.50747145 -0.53296822 -1.72495884
  0.21424725  0.38717894]
Least-squares c (ignoring censored data): [ 3.56700236 -3.01779372 -3.87229159  2.99137944
  3.81418558 -0.9318051   5.55814824  3.45289109 -1.59662757  0.10209278
  3.28915038 -3.57061742  5.59915388  1.08402325  2.59145296 -1.83069947
  0.69343066 -1.45265853]
Relative error (with censored data): 3.2437023515882357
Relative error (least-squares): 19.58042942126819
```

Problem 3: Robust Logistic Regression (Exercise 6.29 in the CVX Book extended exercises)

We are given a data set $\mathbf{x}_i \in \mathbb{R}^d$, $y_i \in \{-1, 1\}, i = 1, \cdots, n$.

We seek a prediction model $\hat{y} = \text{sign}(\theta^T \mathbf{x})$, where $\theta \in \mathbb{R}^d$ is the model parameter.

In logistic regression, θ is chosen as the minimizer of the logistic loss:

$$l(\theta) = \sum_{i=1}^{n} \log\left(1 + \exp\left(-y_i \theta^T \mathbf{x}_i\right)\right)$$

which is a convex function of θ. Here $\|\delta_i\|_\infty = \max_j |\delta_{ij}|$. Remember that each δ_i is a vector with length the same as $\mathbf{x}_i$.

In robust regression, we take into account the idea that the feature vectors $\mathbf{x}_i$ are not known precisely.

Specifically we imagine that each entry of each feature vector can vary by $\pm\epsilon$, where $\epsilon > 0$ is a given uncertainty level.

We define the worst-case logistic loss as:

$$l_{wc}(\theta) = \sum_{i=1}^{n} \sup_{\|\delta_i\|_\infty \leq \epsilon} \log\left(1 + \exp\left(-y_i \theta^T \left(\mathbf{x}_i + \delta_i\right)\right)\right)$$

In words: we perturb each feature vector's entries by up to ϵ in such a way as to make the logistic loss as large as possible. Each term is convex, since it is the supremum of a family of convex functions of θ, and so $l_{wc}(\theta)$ is a convex function of θ.

In robust logistic regression, we choose θ to minimize $l_{wc}(\theta)$.

(a) Explain how to carry out robust logistic regression by solving a single convex optimization problem in disciplined convex programming (DCP) form. Justify any change of variables or introduction of new variables. Explain why solving the problem you propose also solves the robust logistic regression problem.

Hint: $log(1 + exp(u)))$ is monotonic in u.

(b) Fit a standard logistic regression model (i.e., minimize $l(\theta)$), and also a robust logistic regression model (i.e., minimize $l_{wc}(\theta)$), using the data given in rob_regression.csv and rob_regression_test.csv.

The $\mathbf{x}_i$s are provided as the rows of an $n \times d$ matrix named X (these are the variables of the data frame named "X_1, X_2, …"). The y_is are provided as the entries of a n-vector named y (the first column in the data frame).

The file also contains a test data set, X_{test}, y_{test}. Give the test error rate (i.e., fraction of test set data points for which $\hat{y} = y$) for the logistic regression and robust logistic regression models.

```
# load dependencies
import pandas as pd
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
import cvxpy as cp

# load train data
train_url = 'https://media.githubusercontent.com/media/georgehagstrom/DATA609Spring2025/refs/
train_data = pd.read_csv(train_url, index_col='Unnamed: 0')

# load test data
test_url = 'https://media.githubusercontent.com/media/georgehagstrom/DATA609Spring2025/refs/
test_data = pd.read_csv(test_url, index_col = 'Unnamed: 0')

# slice both train and test data
X_train = train_data.drop('y', axis=1).values
y_train = train_data['y'].values
X_test = test_data.drop('y_test', axis = 1).values
y_test = test_data['y_test'].values

# define the standard Logistic Regression
logreg = LogisticRegression(solver='liblinear')
logreg.fit(X_train, y_train)
```

```python
# calculate the prediction and the error rate for Logistic Regression
y_pred_logreg = logreg.predict(X_test)
error_rate_logreg = np.mean(y_pred_logreg != y_test)

accuracy = accuracy_score(y_test, y_pred_logreg)
print(f"Standard Logistic Regression Test Accuracy: {accuracy}")
```

```
Standard Logistic Regression Test Accuracy: 0.8
```