532-Assignment08-Bresser

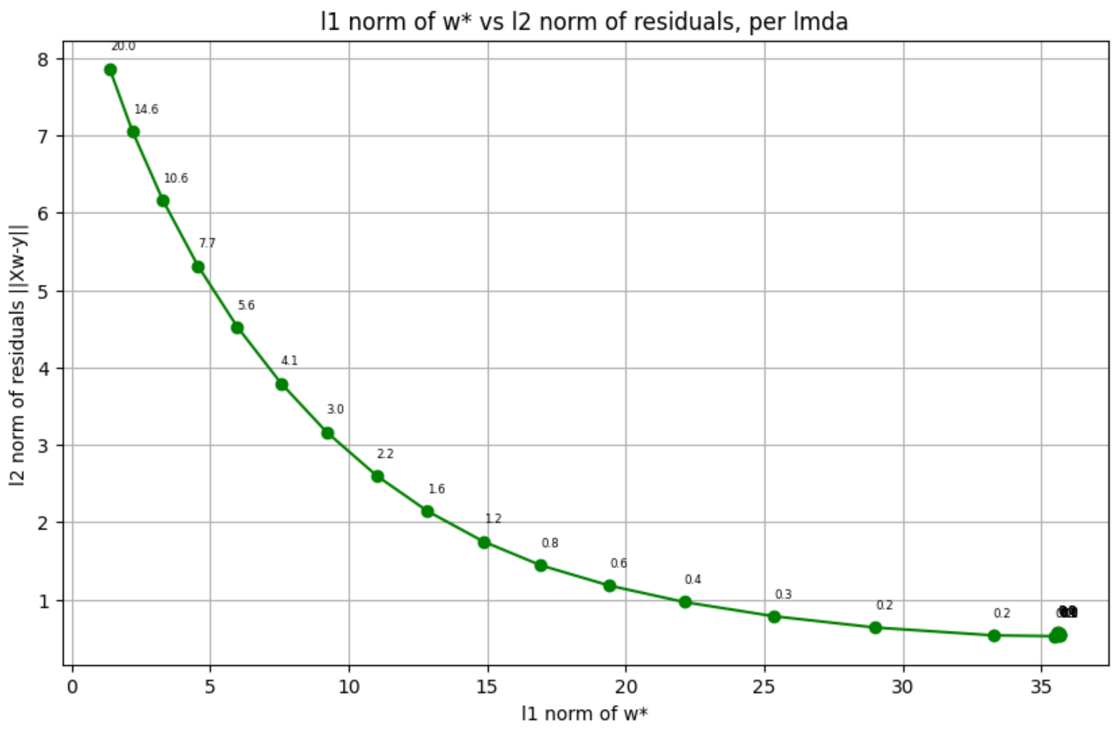
November 20, 2023

[1]:
```
## Breast Cancer LASSO Exploration
## Prepare workspace
from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt
X = loadmat("BreastCancer.mat")['X']
y = loadmat("BreastCancer.mat")['y']
```

[2]:
```
def w_mins_lasso( X_, y_, lmdas_ ):
    # ista_solve_hot: Iterative soft-thresholding for multiple values of
    # lambda with hot start for each case - the converged value for the previous
    # value of lambda is used as an initial condition for the current lambda.
    # this function solves the minimization problem
    # Minimize |Ax-d|_2^2 + lambda*|x|_1 (Lasso regression)
    # using iterative soft-thresholding.
    max_iter = 10**4
    tol = 10**(-3)
    tau = 1/np.linalg.norm(X_,2)**2
    n = X_.shape[1]
    w = np.zeros((n,1))
    num_lam = len(lmdas_)
    X = np.zeros((n, num_lam))
    for i, each_lambda in enumerate(lmdas_):
        for j in range(max_iter):
            z = w - tau*(X_.T@(X_@w-y_))
            w_old = w
            w = np.sign(z) * np.clip(np.abs(z)-tau*each_lambda/2, 0, np.inf)
            X[:, i:i+1] = w
            if np.linalg.norm(w - w_old) < tol:
                break
    return X
```

[65]:
```
# Problem 1a - Devin Bresser

# Need to solve the minimization problem:
# min_w |Aw-d|_2 ^2 + lmda*|w|_1
# For a range of lambdas, 10^-6 to 20
# The given function, ista_solve_hot( A, d, la_array ) accomplishes this.
# input: training or test data (X, y), w_mins associated with training data,␣
↪desired list of lmdas
def plot_solution_norm_vs_bias(X_, y_, w_mins_, lmdas_, show_):

    # Take the l1-norm of each column in w_mins_trunc:
    w_mins_l1_norm = [np.linalg.norm(w, ord=1) for w in w_mins_.T]

    # Compute the residuals ||(X*w_min - y)||_2 for each w_min
    residuals = (X_ @ w_mins_ - y_)

    # Compute the l2-norm of the residuals, down each column
    residuals_norms = np.linalg.norm(residuals, axis=0)

    # Plotting
    if(show_):
        plt.figure(figsize=(10,6))
        plt.plot(w_mins_l1_norm, residuals_norms, marker='o', color="green")
        plt.xlabel('l1 norm of w*')
        plt.ylabel('l2 norm of residuals ||Xw-y||')
        for i in range(0, len(lmdas_)):
            plt.annotate(f'{lmdas_[i]:.1f}', (w_mins_l1_norm[i],␣
↪residuals_norms[i]),
                             textcoords="offset points", xytext=(0,10), ha='left',␣
↪size="6.25")
        plt.title('l1 norm of w* vs l2 norm of residuals, per lmda')
        plt.grid(True)
        plt.show()

    return
```

```
[66]: # Problem 1a continued

# Define training data per problem
X_train, y_train = X[:100], y[:100]

# Define some lmdas to work with
lmdas = np.logspace(-2, np.log10(20), 25)

# Compute the regularized least-squares solutions per lmda with the given␣
↪function:
w_mins_train_lasso = w_mins_lasso(X_train, y_train, lmdas)

# Call above function to generate the desired plot
plot_solution_norm_vs_bias(X_train, y_train, w_mins_train_lasso, lmdas, True)
```



Problem 1a analysis:

We are solving a regularized least-squares problem of the form: min_w ||Xw-y||_2 ^2 + lmda ||w||_1.

In this problem, we need to select lambda to balance bias (accuracy of the solution) and variance (sensitivity of the solution to minor fluctuations). We know that larger l1-norm of the solution indicates a more complex model that may be overfitting to the training data, so we regularize by selecting lambda to punish complexity and encourage sparse solutions (lower l1-norm -> more feature weights equal to zero).

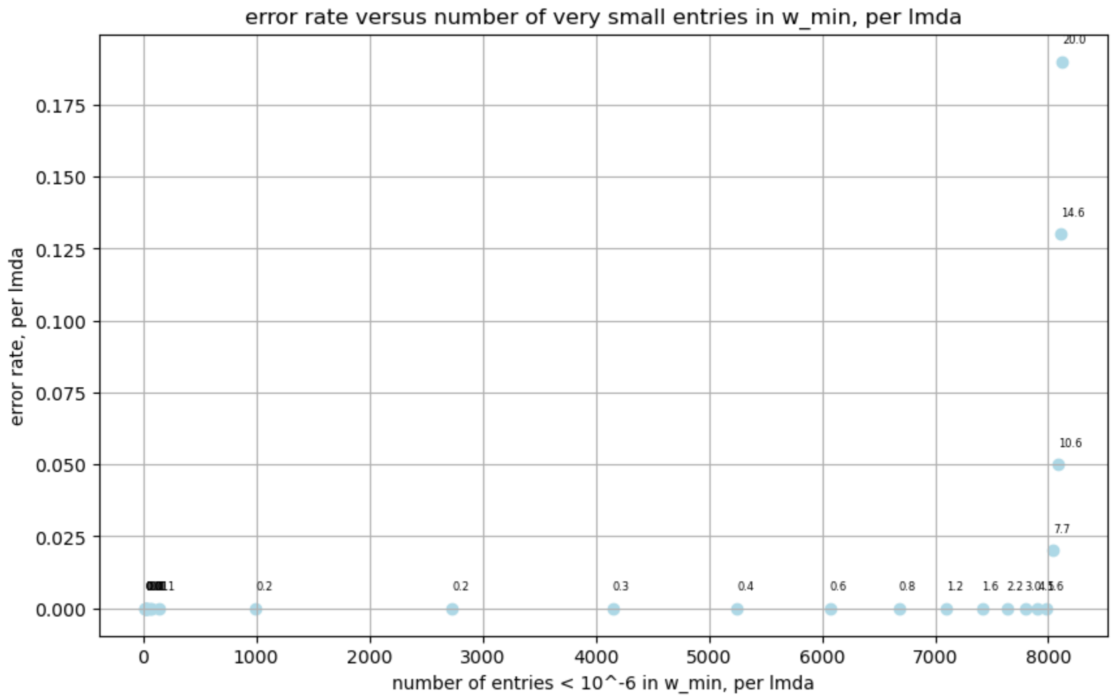
In the above plot, we observe that when when lambda is smaller, the overall solution is closer to the closed-form solution, meaning that the l2-norm of the residuals is quite low (y-axis). But, the l1-norm of the w_min is larger (x-axis), so our model is more complex. This is the expected result. As lambda gets larger, the overall solution deviates from the closed-form solution and the bias increases. But, we get the benefit of a smaller l1-norm solution.

[1]:
```python
# Problem 1b - Devin Bresser

# define a helper method to get errors for each column
def get_error_rates_per_lmda(X_, y_, w_mins_):

    # Create a matrix of predicted values per w_min
    y_raw_matrix = X_ @ w_mins_
    y_pred_matrix = np.sign(y_raw_matrix)
    # Compare each column of y_pred_matrix with y_trunc to compute error rate
    error_rates_per_lmda = np.mean(y_pred_matrix != y_, axis=0)

    # Compute raw squared error per lmda value
    sq_error_per_lmda = np.sum((y_raw_matrix-y_)**2, axis=0)

    return [error_rates_per_lmda, sq_error_per_lmda]
```

[8]:
```
# Problem 1b continued

# input: training or test data (X, y), w_mins associated with training data,␣
 ↪desired list of lmdas
def plot_error_rate_vs_sparsity(X_, y_, w_mins_, lmdas_, show_):

    # Call helper method to compute error rates
    error_rates = get_error_rates_per_lmda(X_, y_, w_mins_)[0]

    # Compute the sparsity for each w_min:
    threshold = 10**-6
    sparsity_per_wmin = np.sum(np.abs(w_mins_) < threshold, axis=0) # going␣
 ↪down cols, count how many with abs < threshold

    # Plot error rate on vertical axis and sparsity on horizontal axis

    if(show_):
        plt.figure(figsize=(10,6))
        plt.scatter(sparsity_per_wmin, error_rates, marker = "o",␣
 ↪color="lightblue")
        plt.xlabel('number of entries < 10^-6 in w_min, per lmda')
        plt.ylabel('error rate, per lmda')
        for i in range(0, len(lmdas_)):
            plt.annotate(f'{lmdas_[i]:.1f}', (sparsity_per_wmin[i],␣
 ↪error_rates[i]),
                          textcoords="offset points", xytext=(0,10), ha='left',␣
 ↪size="6")
        plt.title('error rate versus number of very small entries in w_min, per␣
 ↪lmda')
        plt.grid(True)
        plt.show()

    return error_rates
```

[9]:
```
# Problem 1b continued

# Same X_train, y_train, w_mins_train_lasso, and lmdas as problem 1a
# Call the second plotting function:
```

```
plot_error_rate_vs_sparsity(X_train, y_train, w_mins_train_lasso, lmdas, True)
```

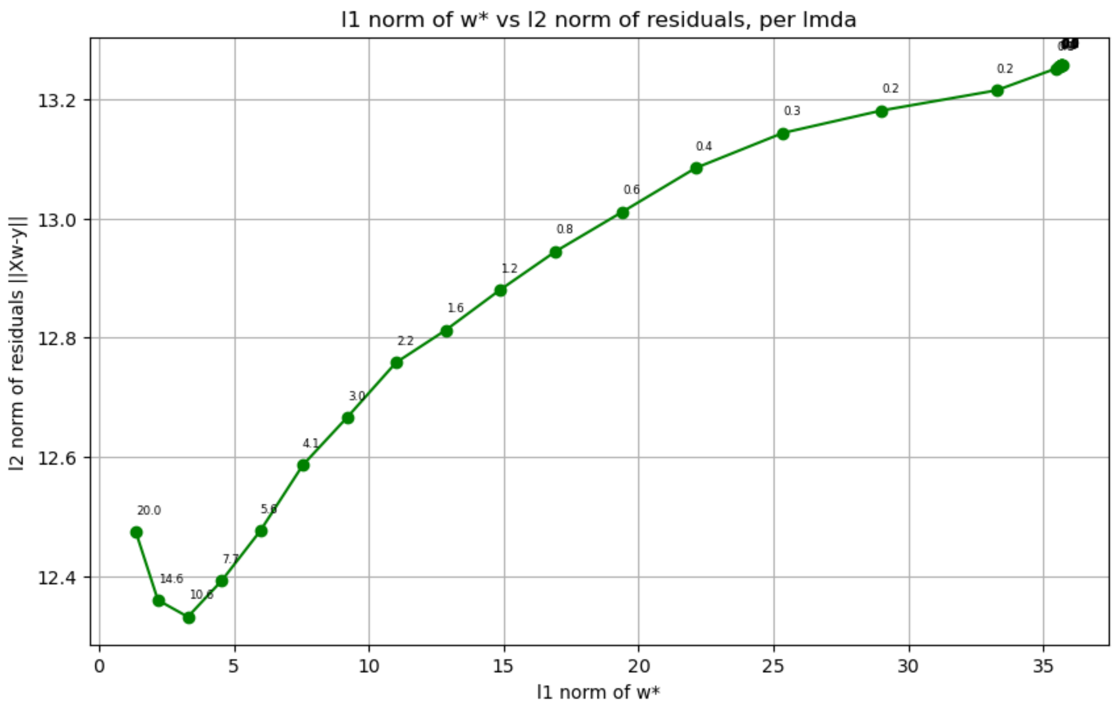


```
[9]: array([0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
        0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.02,
        0.05, 0.13, 0.19])
```

Problem 1b analysis:

As lambda increases, we are more aggressively controlling the norm of the solution, and this is leading to worse accuracy of the solution. As we make lambda very large, we introduce binary misclassifications into the training data, indicating poor performance. But, we get a very sparse solution, meaning the model is less complex and more resistant to noise and errors. The optimal value of lambda clearly lies somewhere between the extremes. It may be a good first choice to pick the largest lambda that leaves 0 training error.

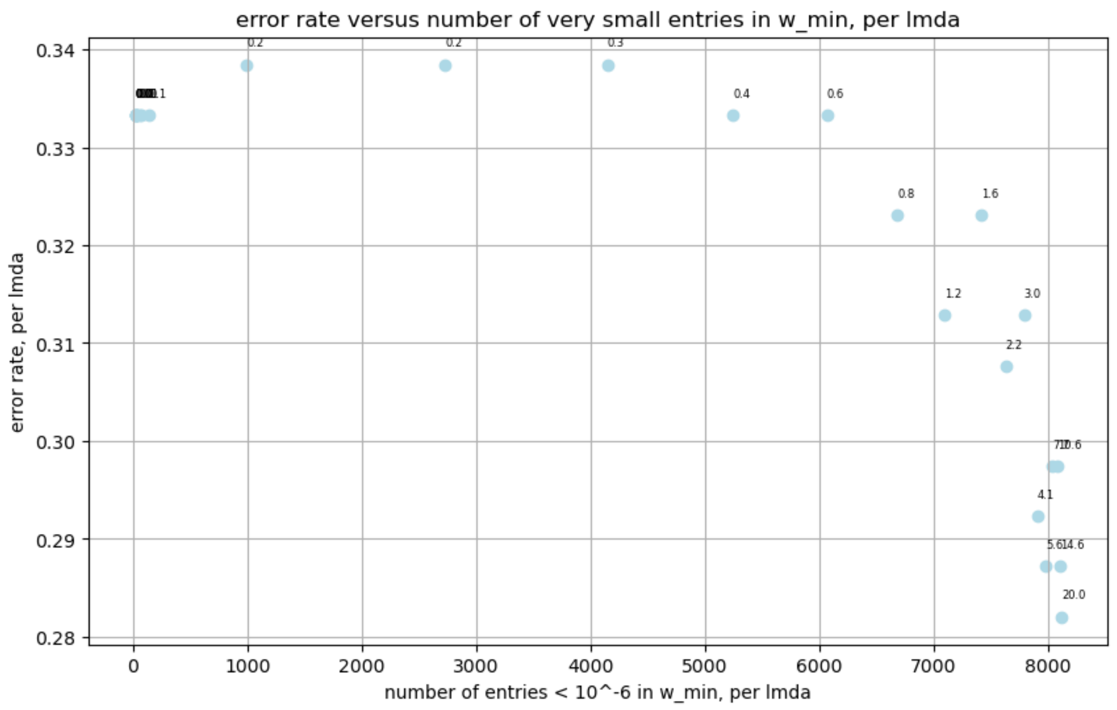
```
[10]: # Problem 1c

# To see how our proposed models perform on new data, we call the functions␣
 ↪from
# before on the test data, with the same w_mins_train_lasso
X_test, y_test = X[100:], y[100:]
plot_solution_norm_vs_bias(X_test, y_test, w_mins_train_lasso, lmdas, True)
```



Problem 1c-i comment:

When we run these proposed models on the test data, we are observing the opposite effect and this is a classic example of overfitting. As lambda decreases, we allow the model to become increasingly complex and fit the training data extremely well (very low residuals ||Xw-y||_2). But, when we apply those complex models to the test data (new, unseen data points), they are too specific and do not perform well.

[11]:
```
# Problem 1c continued

# Calling the second plotting function
plot_error_rate_vs_sparsity(X_test, y_test, w_mins_train_lasso, lmdas, True)
```



```
[11]: array([0.33333333, 0.33333333, 0.33333333, 0.33333333, 0.33333333,
             0.33333333, 0.33333333, 0.33333333, 0.33333333, 0.33846154,
             0.33846154, 0.33846154, 0.33333333, 0.33333333, 0.32307692,
             0.31282051, 0.32307692, 0.30769231, 0.31282051, 0.29230769,
             0.28717949, 0.2974359 , 0.2974359 , 0.28717949, 0.28205128])
```

Problem 1c-ii comment:

We observe similar results to the above plot. When lambda is very small in this case, we have a relatively un-sparse solution, and great performance on the training data. But, in the test data, we are encountering the highest errors, because we have overfit the model. From this set of regularization parameters, it seems like the lowest value of error is attained when lambda is set to 20, with an error rate of 0.282. Going back to problem 1a, we observe that this high value of lambda actually causes the poorest performance of the model on the training data. This counter-intuitive result indicates that the training data has too many features, and many of them are irrelevant to the classification task at hand. Thus, pruning many of them leads to more robust performance on new data.

```
[12]: # Problem 2
      # Define solution to l2-regularized LS problem, per activity 17
      # Note that we should use the form w_min = X^T (X X^T + lambda*I)^-1 * y
      # Because X X^T in this case is 295x295 (or something smaller depending on CV),␣
       ↪whereas X^T X is 8141x8141.

      def w_mins_ridge(X_, y_, lmdas_):
```

```
    w_mins = []
    XXT = X_ @ X_.T
    eye_shape = XXT.shape[0]
    for lmda in lmdas_:
        inv_term = np.linalg.inv(XXT + lmda * np.eye(eye_shape))
        w_min = X_.T @ inv_term @ y_
        w_mins.append(w_min)

    # Convert the list of vectors into a matrix where each column is a w_min
    w_mins_matrix = np.column_stack(w_mins)

    return w_mins_matrix
```

[14]:
```
# Problem 2 - Devin Bresser

# Split the data into training and testing splits per problem

from sklearn.model_selection import KFold

# Let's redefine lmdas to have some more values as we aren't plotting anymore
lmdas = np.logspace(-6, np.log10(20), num=100)

n = 10 # number of CV splits
kf = KFold(n_splits=n, shuffle=True, random_state=42)
splits = kf.split(X)
folds = list(kf.split(X))
X_subsets = [X[idxs] for _, idxs in splits]
# y_subsets = [y[idxs] for _, idxs in splits]
for i in range(n):
    print(X_subsets[i].shape)
```

```
(30, 8141)
(30, 8141)
(30, 8141)
(30, 8141)
(30, 8141)
(29, 8141)
(29, 8141)
(29, 8141)
(29, 8141)
(29, 8141)
```

[ ]:
```
# 1. Use 8 of 10 subsets to compute w_mins_ridge_train and w_mins_lasso_train
# 2. Compute the prediction error on X_test and y_test for each, (X_test and␣
 ↪y_test are ONE of the holdout sets)
# 3. Select the value of lmda that causes the lowest prediction error,␣
 ↪lmda_opt_lasso & lmda_opt_ridge
```

```
# 4. Compute the squared error AND error rate on the OTHER holdout set, using␣
↪lmda_opt_both
# 5. Repeat for all sets of 8/10 subsets, one-testing subset
# 6. Compute the average squared error and average error rate across all␣
↪different subsets, for lasso and ridge.
```

[92]:
```
# Problem 2 continued

from itertools import combinations
import numpy as np
from sklearn.model_selection import KFold

# Initialize the error rate lists
test_error_rates_lasso = []
test_error_rates_ridge = []
squared_errors_lasso = []
squared_errors_ridge = []

# Define the lambda values
lmdas = np.logspace(-6, np.log10(20), num=100)

# create the KFold object with 10 splits
k = 10
kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Generate all possible unique combinations of 8 out of 10 folds for training
fold_indices = list(range(k))
train_combinations = list(combinations(fold_indices, 8))

# Iterate over each combination to perform training, validation, and testing
# Ex: train_indices: [0,1,2,3,4,5,6,7]
for train_indices in train_combinations:

    # The remaining two indices are for validation and test sets
    remaining_indices = list(set(fold_indices) - set(train_indices)) # Ex: [8,9]

    # Run another little loop to be able to do both orders ([val, training] and␣
↪[training, val])
    for a in range(2):

        if(a == 0): val_index, test_index = remaining_indices # Ex: 8, 9
        if(a == 1): test_index, val_index = remaining_indices # Ex: 9, 8

        # Define the training, validation, and test sets
        # String together all of the X & y values with indices in train_indices
        X_train = np.concatenate([X[folds[i][1]] for i in train_indices])
        y_train = np.concatenate([y[folds[i][1]] for i in train_indices])
        X_val = X[folds[val_index][1]] # filter X to indices in the first␣
↪remaining set
        y_val = y[folds[val_index][1]] # filter y to indices in the first␣
↪remaining set
        X_test = X[folds[test_index][1]] # filter X to indices in the second␣
↪remaining set
        y_test = y[folds[test_index][1]] # filter y to indices in the second␣
↪remaining set

        # Compute w_lmda_lasso and w_lmda_ridge on the training set
        w_lmda_lasso = w_mins_lasso(X_train, y_train, lmdas)
        w_lmda_ridge = w_mins_ridge(X_train, y_train, lmdas)

        # Compute the 0/1 errors on the validation set
        val_error_rates_lasso = get_error_rates_per_lmda(X_val, y_val,␣
↪w_lmda_lasso)[0]
        val_error_rates_ridge = get_error_rates_per_lmda(X_val, y_val,␣
↪w_lmda_ridge)[0]

        # Get index of lmda that led to the lowest error rate on the validation␣
↪set
        lmda_opt_lasso_index = np.argmin(val_error_rates_lasso)
        lmda_opt_ridge_index = np.argmin(val_error_rates_ridge)

        # Select the w_lmda associated with the best lambda's index
        w_min_lasso_opt = w_lmda_lasso[:, lmda_opt_lasso_index].reshape(-1, 1)
        w_min_ridge_opt = w_lmda_ridge[:, lmda_opt_ridge_index].reshape(-1, 1)

        # Compute the test error on the test set using the weights associated␣
↪with the best lambda
        [test_error_rate_lasso, squared_err_lasso] =␣
↪get_error_rates_per_lmda(X_test, y_test, w_min_lasso_opt)
        [test_error_rate_ridge, squared_err_ridge] =␣
↪get_error_rates_per_lmda(X_test, y_test, w_min_ridge_opt)

        # Append the error rates to the lists
        test_error_rates_lasso.append(test_error_rate_lasso)
        test_error_rates_ridge.append(test_error_rate_ridge)
        squared_errors_lasso.append(squared_error_rate_lasso)
        squared_errors_ridge.append(squared_error_rate_ridge)
```

[97]:
```
# Problem 2 continued
print(f"Average prediction error rates:\nLASSO: {np.
↪mean(test_error_rates_lasso)} \nRidge: {np.mean(test_error_rates_ridge)}\n")
print(f"Average sum of squared errors:\nLASSO: {np.mean(squared_errors_lasso)}␣
↪\nRidge: {np.mean(squared_errors_ridge)}")
```

```
Average prediction error rates:
LASSO: 0.2979948914431673
Ridge: 0.30288633461047254

Average sum of squared errors:
LASSO: 18.858973871827512
Ridge: 24.62568367982513
```

Conclusion: the LASSO classification method performs slightly better in terms of error rate, and considerably better in terms of raw squared error.

[ ]:

```
```