ista_solve_hot

April 18, 2022

[54]:
```python
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
from sklearn.svm import LinearSVC
import numpy.random
import math
```

[151]:
```python
def ista_solve_hot( A, d, la_array ):
    # 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(A,2)**2
    n = A.shape[1]
    w = np.zeros((n,1))
    num_lam = len(la_array)
    X = np.zeros((n, num_lam))
    for i, each_lambda in enumerate(la_array):
        for j in range(max_iter):
            z = w - tau*(A.T@(A@w-d))
            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
```

[242]:
```python
in_data = loadmat('BreastCancer.mat')
X = in_data['X']
y = in_data['y']
```

[243]:
```python
print(X.shape, y.shape)
```

```
(295, 8141) (295, 1)
```

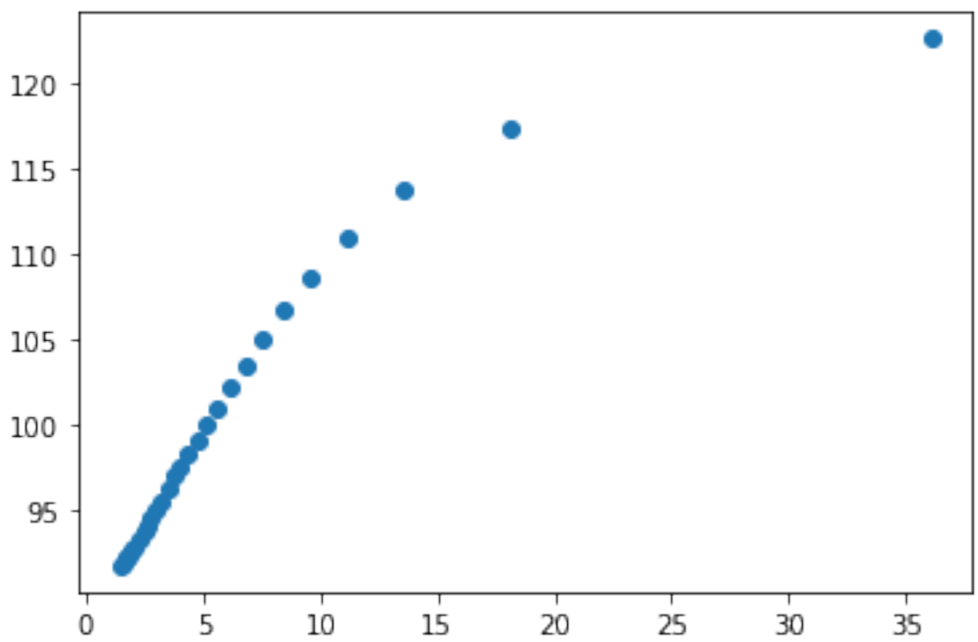
a)

[254]:
```
# We will use the first 100 patients
X_train = X[:100]
y_train = y[:100]
# Set lambda array with range (1e-6, 20) spaced logaritmically
lambda_array = []
i = 1e-6
while i < 19:
    lambda_array.append(i)
    i += math.log(2)
lambda_array.append(20)
w = ista_solve_hot(X_train, y_train, lambda_array)
```

[245]:
```
residual_error = []
w_norm = []
for i in range(len(lambda_array)):
    residual_error.append(np.linalg.norm(X_train@w[:,i] - y_train))
    w_norm.append(abs(w[:,i]).sum())
    print("w_norm: " , w_norm[i],"\tresidual_error: ", residual_error[i],␣
 ↪"\tlambda: ", lambda_array[i])
plt.scatter(x=w_norm, y=residual_error)
plt.show()
# As lambda increases the residual error and the norm of w decreases
```

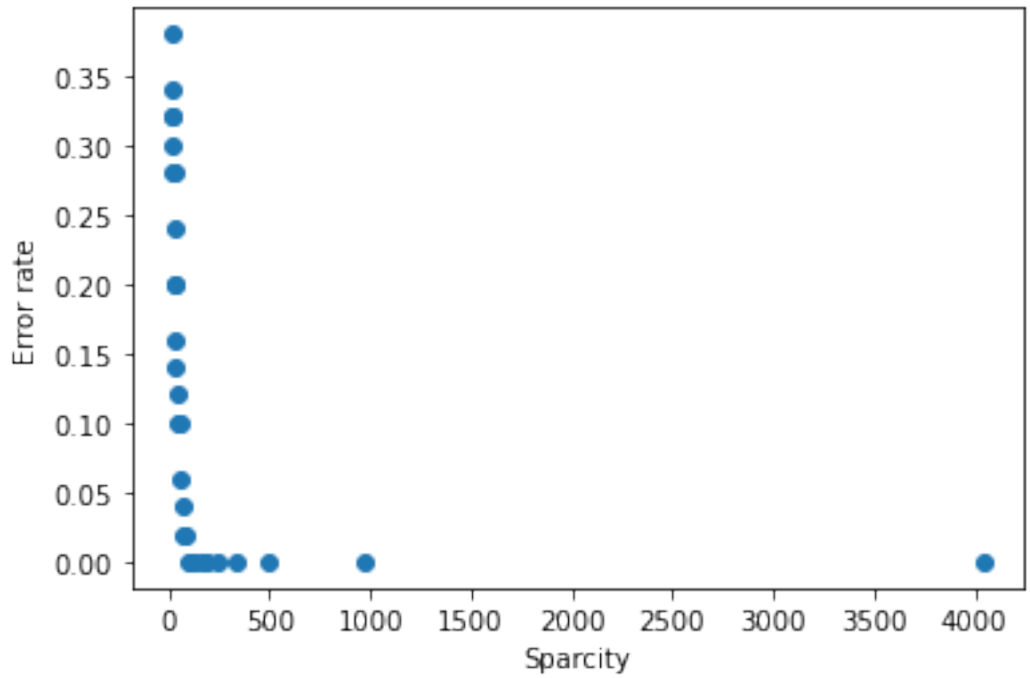
```
w_norm:  36.109055952630406     residual_error:  122.62017023893092     lambda:
1e-06
w_norm:  18.1266980593891       residual_error:  117.3028693792493      lambda:
0.6931481805599453
w_norm:  13.51150995902426      residual_error:  113.75422580631017     lambda:
1.3862953611198905
w_norm:  11.207473857087024     residual_error:  110.96552155127395     lambda:
2.079442541679836
w_norm:  9.600042255143801      residual_error:  108.64335521508443     lambda:
2.7725897222397813
w_norm:  8.425137987735571      residual_error:  106.67098689091421     lambda:
3.4657369027997267
w_norm:  7.504537201776889      residual_error:  104.93988316904763     lambda:
4.158884083359672
w_norm:  6.785326085022339      residual_error:  103.44476518158844     lambda:
4.852031263919617
w_norm:  6.123578969469852      residual_error:  102.12035118812786     lambda:
5.545178444479562
w_norm:  5.579710963832548      residual_error:  100.97384444664706     lambda:
6.238325625039508
w_norm:  5.137386633432994      residual_error:  99.97513990804859      lambda:
6.931472805599453
w_norm:  4.7294810562842615     residual_error:  99.12557162414309      lambda:
7.624619986159399
```

```
w_norm:  4.355999922114707      residual_error:  98.32365768967908      lambda:
8.317767166719344
w_norm:  3.9723493979938773     residual_error:  97.50426863953899      lambda:
9.01091434727929
w_norm:  3.75616629710407       residual_error:  97.0082963354693       lambda:
9.704061527839235
w_norm:  3.4778926816773117     residual_error:  96.27848306896577      lambda:
10.39720870839918
w_norm:  3.136669482594097      residual_error:  95.45134190969435      lambda:
11.090355888959126
w_norm:  2.9903158145972366     residual_error:  95.0538750806501       lambda:
11.783503069519071
w_norm:  2.764883558007083      residual_error:  94.46313591228582      lambda:
12.476650250079016
w_norm:  2.5863002243014237     residual_error:  93.99688909430621      lambda:
13.169797430638962
w_norm:  2.4504933690532065     residual_error:  93.6815802149507       lambda:
13.862944611198907
w_norm:  2.2633392534203525     residual_error:  93.24543582959832      lambda:
14.556091791758853
w_norm:  2.0540384197405395     residual_error:  92.78511251484086      lambda:
15.249238972318798
w_norm:  1.956318032917396      residual_error:  92.6079892263537       lambda:
15.942386152878743
w_norm:  1.8420240652914215     residual_error:  92.39630102542934      lambda:
16.635533333438687
w_norm:  1.7578344358851319     residual_error:  92.2403313247562       lambda:
17.32868051399863
w_norm:  1.6912694674229027     residual_error:  92.11599146101157      lambda:
18.021827694558574
w_norm:  1.5822541740103822     residual_error:  91.92717363184512      lambda:
18.714974875118518
w_norm:  1.426055638400412      residual_error:  91.72082256986741      lambda:
20
```



b)

[246]:
```
error_rate = []
sparcity = np.zeros(len(lambda_array))
for i in range(len(lambda_array)):
    diff = abs(np.sign(X_train@w[:,i]) - y_train[:,0])
    for j in w[:,i]:
        # Add to count the number of nonzero entries of w => if |w|_i > 1e-6
        if j > 1e-6:
            sparcity[i] += 1
    error_rate.append(diff.sum()/len(diff))
plt.scatter(x=sparcity, y=error_rate)
plt.xlabel("Sparcity")
plt.ylabel("Error rate")
plt.show()
# As the number of nonzero entries in w increases => the error rate decreases
```



c)

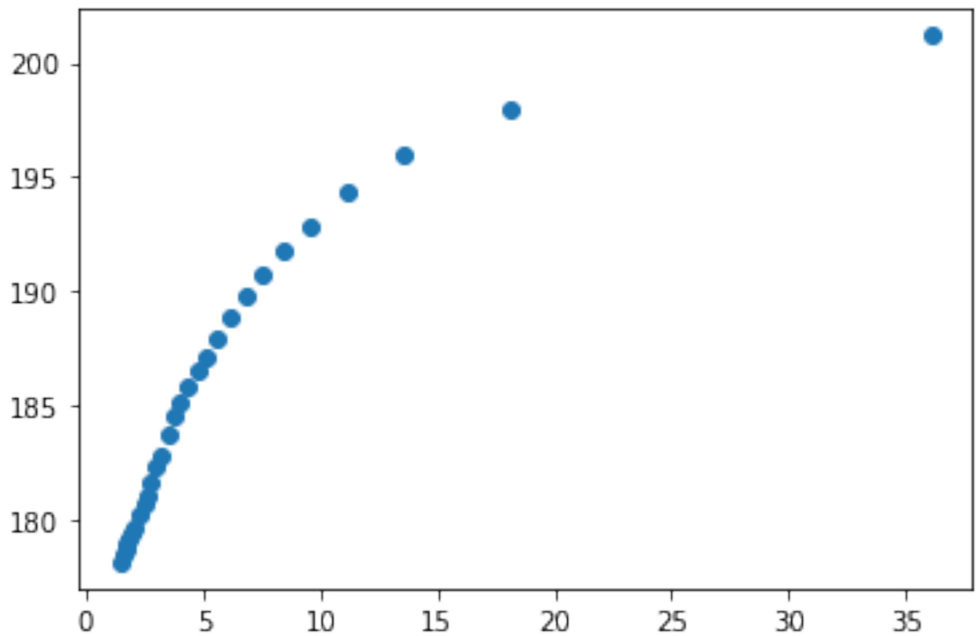
[247]:
```
X_test = X[101:]
y_test = y[101:]
```

[253]:
```
# a)
residual_error = []
w_norm = []
for i in range(len(lambda_array)):
    residual_error.append(np.linalg.norm(X_test@w[:,i] - y_test))
    w_norm.append(abs(w[:,i]).sum())
    print("w_norm: " , w_norm[i],"\tresidual_error: ", residual_error[i],␣
↪"\tlambda: ", lambda_array[i])
plt.scatter(x=w_norm, y=residual_error)
plt.show()
# As with the train data, as lambda increases the residual error and the norm-1␣
↪of w decreases.
# However, the plot show a more curved graph with it's point with a more␣
↪distance between them
```

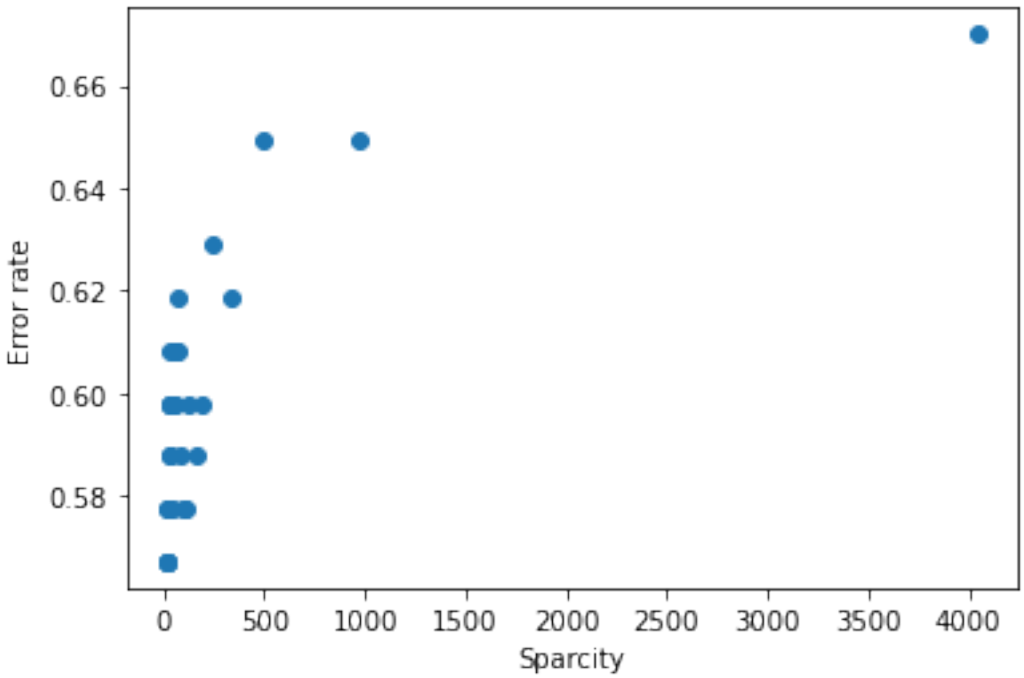
```
w_norm:  36.109055952630406     residual_error:  201.21304378897253     lambda:
1e-06
w_norm:  18.1266980593891       residual_error:  197.91570675411856     lambda:
0.6931481805599453
w_norm:  13.51150995902426      residual_error:  195.93637747867436     lambda:
1.3862953611198905
```

```
w_norm:  11.207473857087024     residual_error:  194.34688419042368     lambda:
2.079442541679836
w_norm:  9.600042255143801      residual_error:  192.82475715297167     lambda:
2.7725897222397813
w_norm:  8.425137987735571      residual_error:  191.78903343197146     lambda:
3.4657369027997267
w_norm:  7.504537201776889      residual_error:  190.782753642679       lambda:
4.158884083359672
w_norm:  6.785326085022339      residual_error:  189.77979622919315     lambda:
4.852031263919617
w_norm:  6.123578969469852      residual_error:  188.87130033785922     lambda:
5.545178444479562
w_norm:  5.579710963832548      residual_error:  187.91498643542022     lambda:
6.238325625039508
w_norm:  5.137386633432994      residual_error:  187.08197273897952     lambda:
6.931472805599453
w_norm:  4.7294810562842615     residual_error:  186.45962974159804     lambda:
7.624619986159399
w_norm:  4.355999922114707      residual_error:  185.78010100452067     lambda:
8.317767166719344
w_norm:  3.9723493979938773     residual_error:  185.0836551926895      lambda:
9.01091434727929
w_norm:  3.75616629710407       residual_error:  184.57120572253032     lambda:
9.704061527839235
w_norm:  3.4778926816773117     residual_error:  183.74603478027777     lambda:
10.39720870839918
w_norm:  3.136669482594097      residual_error:  182.7934822301498      lambda:
11.090355888959126
w_norm:  2.9903158145972366     residual_error:  182.30658498752715     lambda:
11.783503069519071
w_norm:  2.764883558007083      residual_error:  181.60219752866593     lambda:
12.476650250079016
w_norm:  2.5863002243014237     residual_error:  181.06824398874377     lambda:
13.169797430638962
w_norm:  2.4504933690532065     residual_error:  180.7230201718312      lambda:
13.862944611198907
w_norm:  2.2633392534203525     residual_error:  180.17953711686815     lambda:
14.556091791758853
w_norm:  2.0540384197405395     residual_error:  179.62115975839671     lambda:
15.249238972318798
w_norm:  1.956318032917396      residual_error:  179.41276239881375     lambda:
15.942386152878743
w_norm:  1.8420240652914215     residual_error:  179.11612841938015     lambda:
16.635533333438687
w_norm:  1.7578344358851319     residual_error:  178.87761119660146     lambda:
17.32868051399863
w_norm:  1.6912694674229027     residual_error:  178.69479609154843     lambda:
18.021827694558574
```

```
w_norm:  1.5822541740103822     residual_error:  178.4147295472156      lambda:
18.714974875118518
w_norm:  1.426055638400412      residual_error:  178.11348094047474     lambda:
20
```



[252]:
```python
# b)
error_rate = []
sparcity = np.zeros(len(lambda_array))
for i in range(len(lambda_array)):
    diff = abs(np.sign(X_test@w[:,i]) - y_test[:,0])
    for j in w[:,i]:
        # Add to count the number of nonzero entries of w => if |w|_i > 1e-6
        if j > 1e-6:
            sparcity[i] += 1
    error_rate.append(diff.sum()/len(diff))
plt.scatter(x=sparcity, y=error_rate)
plt.xlabel("Sparcity")
plt.ylabel("Error rate")
plt.show()
# Now, the error rate vs sparcity behaves completely different as before. As␣
 ↪the sparcity increases, the error
# rate increases with it.
```


0.66
0.64
0.62
0.60
0.58
Error rate
0
500
1000
1500
2000
2500
3000
3500
4000
Sparcity

Q2_starter

April 18, 2022

[7]:

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

[8]:

```
def ista_solve_hot( A, d, la_array ):
    # 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(A,2)**2
    n = A.shape[1]
    w = np.zeros((n,1))
    num_lam = len(la_array)
    X = np.zeros((n, num_lam))
    for i, each_lambda in enumerate(la_array):
        for j in range(max_iter):
            z = w - tau*(A.T@(A@w-d))
            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
lam_vals = []
i = 1e-6
while i < 19:
    lam_vals.append(i)
    i += math.log(2)
lam_vals.append(20)
```

[78]:
```
##  10-fold CV

# each row of setindices denotes the starting an ending index for one
# partition of the data: 5 sets of 30 samples and 5 sets of 29 samples
setindices =␣
↪[[1,30],[31,60],[61,90],[91,120],[121,150],[151,179],[180,208],[209,237],[238,266],[267,295]

# each row of holdoutindices denotes the partitions that are held out from
# the training set
holdoutindices = [[1,2],[2,3],[3,4],[4,5],[5,6],[7,8],[9,10],[10,1]]

cases = len(holdoutindices)

# be sure to initiate the quantities you want to measure before looping
# through the various training, validation, and test partitions
#
#
#
errors = []
W = []

# Loop over various cases
for j in range(cases):
    # row indices of first validation set
    v1_ind = np.
↪arange(setindices[holdoutindices[j][0]-1][0]-1,setindices[holdoutindices[j][0]-1][1])

    # row indices of second validation set
    v2_ind = np.
↪arange(setindices[holdoutindices[j][1]-1][0]-1,setindices[holdoutindices[j][1]-1][1])

    # row indices of training set
    trn_ind = list(set(range(295))-set(v1_ind)-set(v2_ind))

    # define matrix of features and labels corresponding to first
    # validation set
    Av1 = X[v1_ind,:]
    bv1 = y[v1_ind]

    # define matrix of features and labels corresponding to second
    # validation set
    Av2 = X[v2_ind,:]
    bv2 = y[v2_ind]

    # define matrix of features and labels corresponding to the
    # training set
    At = X[trn_ind,:]
```

```
    bt = y[trn_ind]

#     print(len(v1_ind), len(v2_ind), len(trn_ind))
    w_case = ista_solve_hot(At,bt,lam_vals) # w matrix of w's for each lambda
    error_case = []
    for i in range(len(lam_vals)):
        error_case.append( (np.sign(Av1@w_case[:,i]) - bv1).sum() /len(bv1))
    errors.append(error_case)
    W.append(w_case)

# Use training data to learn classifier
# W = ista_solve_hot(At,bt,lam_vals)
```

[79]:

```
# Find best lambda value using first validation set, then evaluate
# performance on second validation set, and accumulate performance metrics
# over all cases partitions
# We search for the best lambda
min_case_idx = None
min_error = None
min_lambda = None
for i in range(len(errors)): # iterate through each case => i=0 is case of [1,2]
    change_lambda = False
    for j in range(len(errors[i])):
        if min_error == None or errors[i][j] <= min_error:
            min_error = errors[i][j]
            min_lambda = lam_vals[j]
            change_lambda = True
    if change_lambda:
        min_case_idx = i
```

[80]:

```
# Calculate with the best case of houlout and lambda
v1_ind = np.
 ↪arange(setindices[holdoutindices[i][0]-1][0]-1,setindices[holdoutindices[i][0]-1][1])

v2_ind = np.
 ↪arange(setindices[holdoutindices[i][1]-1][0]-1,setindices[holdoutindices[i][1]-1][1])

trn_ind = list(set(range(295))-set(v1_ind)-set(v2_ind))

Av1 = X[v1_ind,:]
bv1 = y[v1_ind]

Av2 = X[v2_ind,:]
bv2 = y[v2_ind]

At = X[trn_ind,:]
bt = y[trn_ind]
```

```
w = ista_solve_hot(At,bt,[min_lambda])

y1_pred = np.sign(Av1@w) - bv1
y2_pred = np.sign(Av2@w) - bv2

y_pred = np.concatenate((y1_pred, y2_pred))

misclassifications = 0
for i in y_pred:
    if i != 0:
        misclassifications += 1

error_rate = misclassifications / len(y_pred)
squared_error = (np.linalg.norm(Av1@w - bv1))**2 + (np.linalg.norm(Av2@w -
 ↪bv2))**2

print("Error rate: " , error_rate)
print("Squared error: ", squared_error)
```

```
Error rate:  0.3898305084745763
Squared error:  55.86164056908563
```

[99]:
```
length = len(At.transpose())
lambdas_matrix = np.zeros((length,length))
np.fill_diagonal(lambdas_matrix, min_lambda)
```

[101]:
```
w_ridge = np.linalg.inv(At.transpose()@At + lambdas_matrix)@At.transpose()@bt
w_ridge
```

[101]:
```
array([[-0.00472367],
       [ 0.00706112],
       [ 0.00881967],
       …,
       [-0.00279695],
       [-0.00814373],
       [-0.00346693]])
```

[105]:
```
y1_pred_ridge = np.sign(Av1@w_ridge) - bv1
y1_pred_ridge = np.sign(Av2@w_ridge) - bv2

y_pred_ridge = np.concatenate((y1_pred_ridge, y1_pred_ridge))

misclassifications_ridge = 0
for i in y_pred_ridge:
    if i != 0:
        misclassifications_ridge += 1
```

```python
error_rate_ridge = misclassifications_ridge / len(y_pred_ridge)
squared_error_ridge = (np.linalg.norm(Av1@w_ridge - bv1))**2 + (np.linalg.
 ↪norm(Av2@w_ridge - bv2))**2

print("Error rate: " , error_rate_ridge)
print("Squared error: ", squared_error_ridge)
```

```
Error rate:  0.2
Squared error:  62.012381589953506
```

We can see that the error rate is lower if we use ridge regression but the squared error is higher

[ ]:

```python

```