Catherine Berrouet, Z23353674

Computational Foundations of AI, Fall 2020

Assignment 1

Task: Ridge Regression Fit

(From Sratch using Python)

You may **not** use a library that can perform *gradient descent, cross validation, ridge regression, least squares regression, optimization, etc.* to successfully complete this programing assignment. The goal of this assignment is not to learn how to use particular libraries of a language, but it is to instead understand how key methods in statistical machine learning are implemented.

Opportunity for 10% extra credit if you additionally implement the assignment using built- in statistical or machine learning libraries (see Deliverable 6 at end of the document).

In this assignment you will be analyzing credit card data from N = 400 training observations. The goal is to fit a model that can predict credit balance based on p = 9 features describing an individual, which include an individual's income, credit limit, credit rating, number of credit cards, age, education level, gender, student status, and marriage status.

Specifically, you will perform a penalized (regularized) least squares fit of a linear model using ridge regression, with the model parameters obtained by batch gradient descent. The tuning parameter will be chosen using five-fold cross validation, and the best-fit model parameters will be inferred on the training dataset conditional on an optimal tuning parameter.

Import Data and Libraries

```
# Import Python libraries for data
import pandas as pd
import numpy as np
# Libraries for plotting
import matplotlib as mpl
import matplotlib.pyplot as plt
# import seaborn as sns


from google.colab import drive
drive.mount('/content/drive')

# Import data from csv file uploaded onto google drive
data = pd.read_csv('/content/drive/My Drive/Colab Notebooks/Computational AI/Credit_N
```

```
#data = pd.read_csv('/content/Credit_N400_p9.csv') #manual upload csv data file
data #prints data for viewing
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dr
```

	Income	Limit	Rating	Cards	Age	Education	Gender	Student	Married	Balan
0	14.891	3606	283	2	34	11	Male	No	Yes	3
1	106.025	6645	483	3	82	15	Female	Yes	Yes	9
2	104.593	7075	514	4	71	11	Male	No	No	5
3	148.924	9504	681	3	36	11	Female	No	No	9
4	55.882	4897	357	2	68	16	Male	No	Yes	3
...	...	...	...	...	...	...	...	...	...	
395	12.096	4100	307	3	32	13	Male	No	Yes	5
396	13.364	3838	296	5	65	17	Male	No	No	4
397	57.872	4171	321	5	67	12	Female	No	Yes	1
398	37.728	2525	192	1	44	13	Male	No	Yes	
399	18.701	5524	415	5	64	7	Female	No	No	9

400 rows × 10 columns

Cleaning and reformat raw data

```
# Reformatting categorical data into numerical binary values
datacopy = data # We use a copy and keep original import
clean = datacopy.replace({'Male': 0, 'Female':1})
clean = clean.replace({'No': 0, 'Yes': 1})
clean
```

	Income	Limit	Rating	Cards	Age	Education	Gender	Student	Married	Balan
0	14.891	3606	283	2	34	11	0	0	1	3
1	106.025	6645	483	3	82	15	1	1	1	9
2	104.593	7075	514	4	71	11	0	0	0	5
3	148.924	9504	681	3	36	11	1	0	0	9

Initialize Variables for Training

Algorithms

Functions

- total_predict: computes predictions
- residual: computes RSS
- cost: computes min total cost
- gradient: computes single gradient
- gradient descent: computes total gradients under max iterations

Summary of Initialized Variables

- Recall our parameters for use in the coded algorithms below: X, Y, N, p, beta
- X is our normalized training data centered and scaled to have unit standard deviation
- Y is the true value data generated as N-dimensional centered response vector $\mathbf{y}$
- N is the number of rows
- p is the number of columns should be = (dimension of beta)
- beta is our random initialized parameter vector

Now we will introduce some additional assumptions for our fitting as follows:

- tune is the Tuning Parameters Vector aka our lambda used in our Cost Function Equation
- iter is set to 1000
- alpha is initialized to 10^(-5) as suggested to act as proof of convergence within 1000 iterations

```
# Training Data, X, Y, N, p
training_data = clean.iloc[:, :-1]                                  # we only take the
X = (training_data - training_data.mean())/training_data.std()     # X_normalized
Y_data = pd.DataFrame(clean.iloc[:, -1])                            # dependent variabl
Y_centered = Y_data - Y_data.mean(axis=0)                           # Y_centered
Y = pd.DataFrame(Y_centered)
N = X.shape[0]                                                      # 400 rows
p = X.shape[1]                                                      # 9 columns
# randomized initialization beta vector
random_initialization = np.random.uniform(low=-1.0, high=1.0, size=p)
```

```
beta = pd.DataFrame(np.random.uniform(low=-1.0, high=1.0, size=p))
# tuning parameter vector
tune = [10**(-2), 10**(-1), 10**(0), 10, 10**(2), 10**(3), 10**(4)]
max_iter = 1000
alpha = 10**(-5)

# Viewing all variables
print('Training Data')
print(training_data)
print('')
print('X')
print(X)
print('Y')
print(Y)
print('')
print('N =', N, ', p =', p)
print('')
print('beta = ', beta)
print('dimension:', beta.shape)
print('')
print('lambda tuning vector = ', tune)
print('')
print('alpha:', alpha)
```

```
Training Data
       Income  Limit  Rating  Cards  Age  Education  Gender  Student  Married
0      14.891   3606     283      2   34         11       0        0        1
1     106.025   6645     483      3   82         15       1        1        1
2     104.593   7075     514      4   71         11       0        0        0
3     148.924   9504     681      3   36         11       1        0        0
4      55.882   4897     357      2   68         16       0        0        1
..        ...    ...     ...    ...  ...        ...     ...      ...      ...
395    12.096   4100     307      3   32         13       0        0        1
396    13.364   3838     296      5   65         17       0        0        0
397    57.872   4171     321      5   67         12       1        0        1
398    37.728   2525     192      1   44         13       0        0        1
399    18.701   5524     415      5   64          7       1        0        0

[400 rows x 9 columns]

X
       Income     Limit    Rating  ...    Gender   Student   Married
0   -0.860505 -0.489386 -0.464957  ... -1.034339 -0.332916  0.794400
1    1.725276  0.827225  0.827667  ...  0.964384  2.996248  0.794400
2    1.684646  1.013518  1.028023  ... -1.034339 -0.332916 -1.255665
3    2.942467  2.065853  2.107363  ...  0.964384 -0.332916 -1.255665
4    0.302549  0.069925  0.013314  ... -1.034339 -0.332916  0.794400
..        ...       ...       ...  ...       ...       ...       ...
395 -0.939809 -0.275366 -0.309842  ... -1.034339 -0.332916  0.794400
396 -0.903832 -0.388875 -0.380936  ... -1.034339 -0.332916 -1.255665
397  0.359012 -0.244606 -0.219358  ...  0.964384 -0.332916  0.794400
398 -0.212542 -0.957716 -1.053100  ... -1.034339 -0.332916  0.794400
399 -0.752403  0.341565  0.388175  ...  0.964384 -0.332916 -1.255665

[400 rows x 9 columns]
Y
     Balance
```

Training

Prediction Function

```
398 -520.015
```

```
Prediction Function
ef total_predict(X, B):
'''
  This function computes the dot product between the rows of design matrix and parame
  It uses the predict function to iterate through each row of design matrix.
  :param X: training_data as pandas dataframe (i.e our Nxp design matrix)
  :param B: is parameters_vector as pandas dataframe syntax (i.e p-dimensional B)
  :return: predictions vector (i.e. dot product of X and beta parameters vector)
'''
    # Corrective measure: Double check appropriate dimensions of dataframes
dim1 = X.shape[1]
dim2 = B.shape[0]
```

```
  if (dim1) == (dim2):
    predictions = X.dot(B.to_numpy())
    return (predictions)
  else:
    print('Cannot compute. Please check Dimensions!')
    print('Dimensions of design matrix Nxp: ', X.shape)
    print('Dimensions of Initialized Parameter Vector: (', len(B), 'x 1)')

  Output
 ıat = total_predict(X, beta)
 :int('Total predictions, yhat:')
 :int(yhat)
 :int('')
 :int('yhat Transposed First few entries:')
 :int((yhat.T).head(3))
```

```
Total predictions, yhat:
            0
0   -0.186870
1    0.398970
2    1.526057
3    4.909429
4   -1.824202
..        ...
395 -0.665781
396 -1.803241
397 -0.053657
398 -0.499326
399  1.667169

[400 rows x 1 columns]

yhat Transposed First few entries:
       0        1         2         3    ...       396       397       398
0 -0.18687  0.39897  1.526057  4.909429  ... -1.803241 -0.053657 -0.499326  1.66

[1 rows x 400 columns]
```

RSS Function

```
# RSS Function
def residual(Y, yhat):
  '''
  :param Y: true values
  :param yhat: predictions (i.e. dot product of X and parametric vector beta)
  :return: residual sum of sqaures
  '''
  dim1 = Y.shape
  dim2 = yhat.shape
  print("Y and yhat Dimensions Check:")
  print(dim1, 'and', dim2)
  rss = pd.DataFrame((Y.values - yhat.values)**2)
```

```
  return rss

# Output
residual_result = residual(Y,yhat)
print('rss:', residual_result)
```

```
Y and yhat Dimensions Check:
(400, 1) and (400, 1)
rss:                   0
0     34904.750080
1    146372.070459
2      3417.448002
3    192787.357378
4     35040.394995
..             ...
395    1652.486000
396    1460.138553
397  145894.467791
398  269896.535752
399  197418.335069

[400 rows x 1 columns]
```

Cost Function

```
# Cost Function
def cost(rss, tuning, b):
  '''
  :param b: parametric_vector_B
  :param rss: residual sum of squares
  :param tuning: tuning_parameter_vector
  :return: minimum cost computation
  '''
  # Checking pandas DataFrame dimensions
  tuning = pd.DataFrame(tuning)
  dim1, dim2, dim3 = rss.shape, tuning.shape, b.shape
  print('')
  print('rss Dimensions Check:', dim1)
  print('tuning parameter vector Dimensions Check:', dim2)
  print('b randomized vector Dimensions Check:', dim3)
  print('')
  # Cost Computation
  tobesummed = []
  for j in range(len(b)):
    compute = tuning @ ((b.iloc[j])**2)
    tobesummed.append(compute)
  regularization = sum(tobesummed)
  total_cost = rss.values + ((regularization).T).values
  # Convert to pandas Dataframe syntax
  return pd.DataFrame(total_cost)
```

```
# Output
rss1 = residual(Y, yhat)
cost_result = cost(rss1, tune, beta)
print('Cost Function Computation:')
print(cost_result)
```

```
Y and yhat Dimensions Check:
(400, 1) and (400, 1)

rss Dimensions Check: (400, 1)
tuning parameter vector Dimensions Check: (7, 1)
b randomized vector Dimensions Check: (9, 1)

Cost Function Computation:
                 0              1  ...              5              6
0     34904.775643   34905.005716  ...   37461.106772   60468.317005
1    146372.096023  146372.326095  ...  148928.427152  171935.637385
2      3417.473565    3417.703638  ...    5973.804694   28981.014927
3    192787.382942  192787.613014  ...  195343.714071  218350.924304
4     35040.420558   35040.650630  ...   37596.751687   60603.961920
..             ...            ...  ...            ...            ...
395    1652.511564    1652.741636  ...    4208.842693   27216.052926
396    1460.164116    1460.394189  ...    4016.495245   27023.705478
397  145894.493355  145894.723427  ...  148450.824484  171458.034717
398  269896.561316  269896.791388  ...  272452.892445  295460.102678
399  197418.360632  197418.590704  ...  199974.691761  222981.901994

[400 rows x 7 columns]
```

Single Gradient Function

```
# Single Gradient Computation Function
def gradient(X,Y, b, tuning_parameter_vector, alpha):
  '''
  This function computes the gradient for each b_j in the parametric vector B.
  :param X: training_data, this is our Nxp standardized matrix
  :param Y: normalized predictions, our yhats
  :param b: randomly initialized parametric vector, beta
  :param tuning_parameter_vector: our lambdas vector of 7 values
  :param alpha: starting point for learning

  Note: This is just 1 iteration to simply test gradient computation.
  '''
  # Comment: I've broken down each step of the computation mathematically
  # in order to ensure the syntax for the pandas Dataframe is correct and precise
  for lambda_value in tuning_parameter_vector:
    for k in range(len(b)):
        step1 = Y.values -(X.dot(b.to_numpy()))
        X_t = X.iloc[:, :k] # X column k transposed
        step2 = (X_t).T  @ step1
        step3 = lambda value * b.iloc[k] - step2
```

```
        step4 = 2 * lambda_value * step3
        step5 = b.iloc[k] - step4
        beta_update = step5
  return beta_update

# Output
print('Single gradient function computation (check):')
singlegradient = gradient(X,Y, beta, tune, alpha)
print(singlegradient)
```

```
Single gradient function computation (check):
                      0
Income     1.818221e+09
Limit      3.279393e+09
Rating     3.286506e+09
Cards      4.405286e+08
Age        1.335926e+08
Education  9.937858e+07
Gender     1.978962e+08
Student    1.071097e+09
```

Gradient Descent Function

```
# Gradient Descent Function
def gd(X,Y,tune,alpha,max_iter):
  '''
  :param X: our standardized Nxp design matrix, <class 'pandas.core.frame.DataFrame>
  :param Y: our N-dimensional centered y
  :param beta: our randomply initialized parametric vector B
  :param max_iter: max iterations alloted for convergence
  :param alpha: proof of when convergence occurs (i.e. 2*alpha*lambda <1)
  :param L: lambda value in tuning parameter vector
  :return beta_update, cost: total gradient calculation and cost computation

  This function is used for 'training' phase/step.
  '''
  # Computations
  b = pd.DataFrame(data=np.random.uniform(-1, 1, X.shape[1]))
  XB = pd.DataFrame((X.values).dot(b.values))
  Y_minus_XB = pd.DataFrame(Y.values - XB.values)
  X_T_dotprod_Y_minus_XB = (X.T).dot(Y_minus_XB)

  # Change to dataframe syntax for computations
  tuning = pd.DataFrame(tune)
  Lb = tuning.dot(b.T)
  # Making pretty dataframe for lambabeta - for faster computations
  d = pd.Series(tune)
  lambda_beta_df = (Lb.T).rename(columns = d, inplace = False)
  # We will use this dataframe to plot a graph later
```

```
    # Iterate for every column in lambda_beta dataframe and subtract XB column
    for i in lambda_beta_df.columns:
      compute = lambda_beta_df[i].values - (X_T_dotprod_Y_minus_XB.iloc[:,0].values)

    # Iterate for 1000 iterations - this yields convergence by given assumption
    for iteration in range(max_iter):
      b_temp = (2*alpha)*(pd.DataFrame(compute))
      b_update = b - b_temp
    return b_update, lambda_beta_df # return new updated beta vector

# Output
gd_computation = (gd(X, Y, tune, alpha, 1000)[0])
to_plot = (gd(X, Y, tune, alpha, 1000)[1])
print('For gradient descent:')
print(gd_computation)
```

```
For gradient descent:
          0
0  1.587045
1  2.522692
2  3.890038
3  0.746434
4 -0.220751
5  0.128189
6 -0.505701
7  0.929778
8  0.762069
```

Deliverable 1

Illustrate the effect of the tuning parameter on the inferred ridge regression coefficients by generating a plot of nine lines (one for each of the p = 9 features), with the y-axis as $\beta_j = 1, 2, \ldots, 9$, and the x-axis the corresponding log-scaled tuning parameter value $\log_{10}(\lambda)$ that generated the particular $\hat{\beta}_j$. Label both axes. Without the log scaling of the tuning parameter, the plot will look distorted.

```
d1 = pd.DataFrame(to_plot.T)
# Adding labels for corresponding b_j values in beta vector computed in dataframe
beta = pd.Series(['b1', 'b2', 'b3', 'b4', 'b5', 'b6', 'b7', 'b8', 'b9'])
d1 = (d1).rename(columns = beta, inplace = False)
# View our dataframe which we will use to plot for Deliverable 1
d1

# Notes for Plotting
# Each column for b_j's in the d1 dataframe will be the plotted y-axis points
# And the x-axis will consist of the lambda values on the log scale (first column of
```

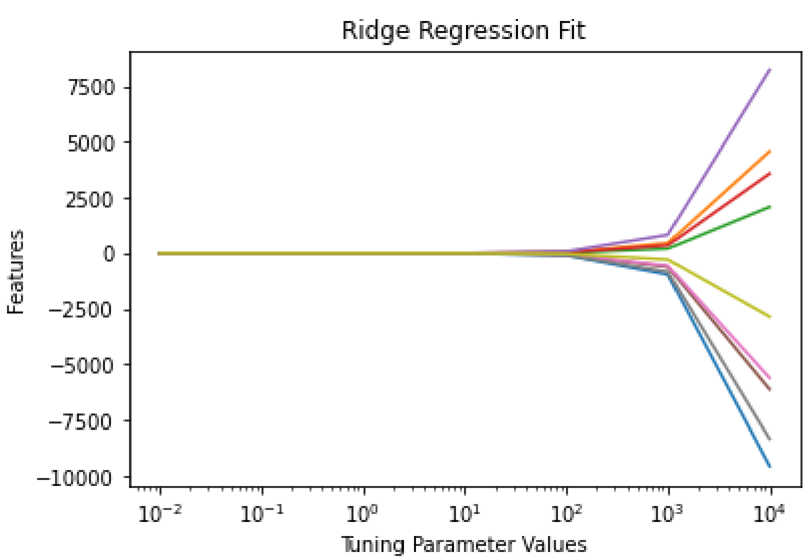
	b1	b2	b3	b4	b5	b6
0.01	-0.009564	0.004572	0.002080	0.003576	0.008227	-0.006119
0.10	-0.095641	0.045722	0.020798	0.035764	0.082268	-0.061185
1.00	-0.956413	0.457224	0.207979	0.357640	0.822679	-0.611854
10.00	-9.564132	4.572236	2.079788	3.576396	8.226792	-6.118544
100.00	-95.641325	45.722357	20.797877	35.763959	82.267915	-61.185443
1000.00	-956.413248	457.223572	207.978773	357.639594	822.679150	-611.854429
10000.00	-9564.132478	4572.235719	2079.787728	3576.395942	8226.791501	-6118.544289

```
# Initialize figure
fig = plt.figure()
plt.title('Ridge Regression Fit')

# Plotting
for i in range(len(d1.columns)):
  # tuning parameter lambda and lambda_beta column values
  plt.plot(tune, d1.iloc[:, i])

# Axes
plt.xscale('log')
plt.xlabel('Tuning Parameter Values')
plt.ylabel('Features')

plt.show()
fig.savefig('Deliverable1.jpg')
```



Task: Cross Validation

Step 1:

First we divide the data into K equal parts. Below I'm going to use the resulting dataframe with 80 rows and 5 columns. Each column is a 'split' aka a fold. We will use the folds for the next step.

Algorithms

Functions

- Cross Validation Split: computes folds
- CV: computes Cross Validation

Summary of Initialized Variables

- X_d2 is the normalized training data centered and scaled to have unit standard deviation
- Y_d2 is the true values data (i.e. Credit Balances) generated as N-dimensional centered response vector $\mathbf{y}$
- K is the number of folds

Same parameters as before repeated again as:

- b_d2 will be our random initialized parameter vector of length = dim k.
- tune is the Tuning Parameters Vector aka our lambda used in our Cost Function Equation
- maximum iteration is 1000
- alpha is initialized to 10^(-5) as suggested to act as proof of convergence within 1000 iterations

Cross Validation Split Function

```
# Set K parameter for K-fold Cross Validation
K = 5

# Import Library for randomization
import random
from random import randrange

# Cross Validation Split Function
def cross_validation_split(dataset, folds= K):
  dataset_split = list()
  dataset_copy = list(dataset)
  fold_size = int(len(dataset) / folds)
  for i in range(folds):
    fold = list()
    while len(fold) < fold_size:
      index = randrange(len(dataset_copy))
      fold.append(dataset_copy.pop(index))
```

```
    dataset_split.append(fold)
  return dataset_split
```

Initialize Variables for Training

```
# Grabbing 5 data folds from X data
random.seed(1)                                      # random seed
                                                    # d2 = (training_data.iloc[:,: -1]).val
                                                    # .values turnes into np array
d2 = X.values
X_folds = cross_validation_split(d2, K)             # cross validation
X_folds_df = (pd.DataFrame(X_folds))                # convert to dataframe
X_folds_df.T                                        # Transposed dataframe
                                                    # Note: fold_size = total_rows / total_
groups = pd.Series(['k1', 'k2', 'k3', 'k4', 'k5'])
X_d2 = (X_folds_df.T).rename(columns = groups, inplace = False)
X_d2
```

	k1	k2	k3	
0	[-0.4928995107556017, 0.3827226588126472, 0.40...	[-0.8363879391677917, 0.02270168363344191, 0.0...	[-0.11899479329187766, -0.21947849100575875, -...	[-0.80886573732 0.233255466951
1	[0.17912456217870565, 0.15613906071550596, 0.1...	[-0.3490747260520339, -1.2778795046794031, -1....	[1.8655262003008712, 0.8345901402950914, 0.730...	[0.0223615052666 0.82375918436

```
# Grab the 5 datafolds for true value Y data
random.seed(1)                                   # random seed
# balance = (clean['Balance']).values          # np array
balance = Y.values
Y_folds = cross_validation_split(balance, K)   # cross validation
Y_folds_df = (pd.DataFrame(Y_folds))         # convert to dataframe
Y_folds_df.T                                 # Transposed dataframe dim: 80 x 5
# Note: there are 80 rows per fold as we expected: fold_size = total_rows / K
Y_d2 = (Y_folds_df.T).rename(columns = groups, inplace = False)
Y_d2
```

	k1	k2	k3	k4
0	[301.985]	[168.985]	[-274.015]	[342.985]
1	[-39.014999999999986]	[-520.015]	[-129.015]	[525.985]
2	[-520.015]	[20.985000000000014]	[-270.015]	[211.985]
3	[5.985000000000014]	[533.985]	[-520.015]	[-101.01499999999999]
4	[391.985]	[582.985]	[8.985000000000014]	[-504.015]
...	...	...	...	...
75	[834.985]	[-520.015]	[-200.015]	[-357.015]
76	[-55.014999999999986]	[241.985]	[-520.015]	[276.985]
77	[224.985]	[424.985]	[-520.015]	[-38.014999999999986]
78	[511.985]	[142.985]	[-520.015]	[291.985]
79	[-520.015]	[-520.015]	[-138.015]	[655.985]

80 rows × 5 columns

Step 2:

Now that we've split the the data into k = 5 groups (columns in Step 1).

So that we now run a 5-fold cross validation for every lambda in our tuning parameter vector and evaluated 5 times using the performance summarized by taking the mean performance score (using our gradient descent).

Cross Validation Function

```
# Cross Validation Algorithm
def CV(X_d2, Y_d2, tune, alpha):
  '''
  Cross Validation Function
  :param X_d2: k-fold data dataframe
  :param Y_d2: true values fold data dataframe
  :param tune: tuning parameter of lambda values
  :param alpha: learning rate
  :return: beta updated dataframe for each fold, cost dataframe for each fold
  '''
  CV_5 = (gd(X_d2, Y_d2, tune, alpha, 1000))
  cv_beta = CV_5[0] # beta_update
  CV_5_beta = (cv_beta).rename(columns = groups, inplace = False)
  CV_5_cost = CV_5[1]
  CV_5_cost = (CV_5_cost.T).rename(columns = groups, inplace = False)
  return CV_5_beta, CV_5_cost

# Output for Viewing
# print('Cross Validation beta updates for each k-fold:')
# print(CV(X_d2, Y_d2, tune, alpha)[0])
# print('')
# print('Cost for each CV fold')
# print(CV(X_d2, Y_d2, tune, alpha)[1])
```

Cross Validation beta updates for each k-fold

```
CV_5 = (gd(X_d2, Y_d2, tune, alpha, 1000)[0]) # beta_update
CV_5_beta = (CV_5.T).rename(columns = groups, inplace = False)
CV_5_beta

# for beta in CV_5_beta['k1'][0]:
#   print(beta)
```

	k1	k2	k3	k4	
0	[0.8665391115655625, 1.1009242364545764, 1.102...	[0.5821675881616334, 0.5780734157082644, 0.579...	[0.44399875622923, 0.4280314240447615, 0.44228...	[-0.3996372925768816, -0.4466357169428901, -0....	[((

Cost for each K-fold

```
CV_5_cost = gd(X_d2, Y_d2, tune, alpha, 1000)[1].T # Cost
CV_5_cost= (CV_5_cost).rename(columns = groups, inplace = False)
CV_5_cost
```

	k1	k2	k3	k4	k5
0.01	-0.007153	-0.008870	-0.006103	-0.005480	-0.006180
0.10	-0.071527	-0.088704	-0.061025	-0.054804	-0.061800
1.00	-0.715267	-0.887038	-0.610250	-0.548038	-0.617999
10.00	-7.152670	-8.870383	-6.102501	-5.480376	-6.179986
100.00	-71.526696	-88.703829	-61.025009	-54.803758	-61.799864
1000.00	-715.266960	-887.038285	-610.250089	-548.037585	-617.998636
10000.00	-7152.669600	-8870.382851	-6102.500892	-5480.375848	-6179.986362

Deliverable 2

Illustrate the effect of the tuning parameter on the cross validation error by generating a plot with the y-axis as $CV_{(5)}$ error, and the x-axis the corresponding log-scaled tuning parameter value $\log_{10}(\lambda)$ that generated the particular $CV_{(5)}$ error. Label both axes. Without the log scaling of the tuning parameter, the $CV_{(5)}$ plot will look distorted.

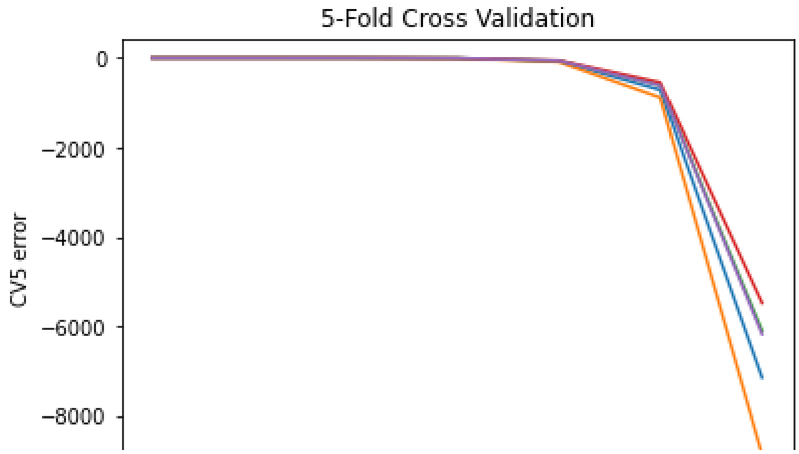
```
# Initialize figure
fig = plt.figure()
plt.title('5-Fold Cross Validation')

# Plotting
for i in range(len(CV_5_cost.columns)):
  # tuning parameter lambda and lambda_beta column values
  plt.plot(tune, CV_5_cost.iloc[:, i])

# Axes
plt.xscale('log')
plt.xlabel('Tuning Parameter Values')
plt.ylabel('CV5 error')

# Legend
# plt.legend(loc="upper left")

plt.show()
fig.savefig('Deliverable1.jpg')
```



Deliverable 3

Indicate the value of λ that generated the smallest $CV_{(5)}$ error.

```
# We need to find the minimum value of all the rows
# Then the row with the smallest set of values,
# will correspond to the lambda which generated the smallest error.

# Get minimum errors list for every fold in Cost dataframe
for row,column in CV_5_cost.iterrows():
  min1 = min(CV_5_cost['k1']) # k1
  min2 = min(CV_5_cost['k2']) # k2
  min3 = min(CV_5_cost['k3']) # k3
  min4 = min(CV_5_cost['k4']) # k4
  min5 = min(CV_5_cost['k5']) # k5
  Total_Min = pd.Series([min1,min2,min3,min4,min5])

# Get minimum error of all folds minimums errors list
print('All Minimums for each fold')
print(Total_Min.values)
min_found =  min(Total_Min)
print('min:', min_found)

# Get lambda Value that corresponds to the minimum error
for i in range(len(Total_Min)):
  if Total_Min[i] == min_found:
    print('index:', i)

# CV_5_cost.iloc[0,i]
lowest_error_lambda = tune[i]
print('Lambda Value Found that generated the smallest CV(5) error was', lowest_error_
import math
print('That is Lambda 10^', math.log10(lowest_error_lambda))
```

```
All Minimums for each fold
[-7152.66960011 -8870.38285108 -6102.50089151 -5480.37584775
 -6179.98636228]
min: -8870.382851082506
index: 1
```

Deliverable 4

Given the optimal λ, retrain your model on the entire dataset of N = 400 observations and provide the estimates of the p = 9 best-fit model parameters.

```
best_lambda = [10**2]
retrain = gd(X,Y,best_lambda,alpha,1000)
print('Retrained with the best lambda value, we get the following gradient descent:')
col = pd.Series(['b1', 'b2','b3','b4','b5','b6','b7','b8', 'b9'])
d4 = pd.DataFrame(retrain[0])
d4 = d4.T
d4.rename(columns = col, inplace = False)
```

```
Retrained with the best lambda value, we get the following gradient descent:
```

	b1	b2	b3	b4	b5	b6	b7	b8	]
0	1.810669	3.847291	2.62477	1.274856	-0.449715	-0.851278	-0.056418	1.836848	-0.7367

Deliverable 5

Provide all your source code that you wrote from scratch to perform all analyses (aside from plotting scripts, which you do not need to turn in) in this assignment, along with instructions on how to compile and run your code.

For the Ridge Regression Fitting, we have the following functions with their titles that compute the following:

1. Prediction function: *total_predict*
2. RSS Function: *residual*
3. Cost Function: *cost*
4. Gradient Descent: *gd*

Under each function are listed the description of each parameter necessary to run the function's algorithm. Underneath each function is also listed the "Output" that is a designated sample instructions/method as on how to use the function.

```
# Source Codes Listed
```

```
# Prediction Function
def total_predict(X, B):
  '''
    This function computes the dot product between the rows of design matrix and parameters
    It uses the predict function to iterate through each row of design matrix.
    :param X: training_data as pandas dataframe (i.e our Nxp design matrix)
    :param B: is parameters_vector as pandas dataframe syntax (i.e p-dimensional B)
    :return: predictions vector (i.e. dot product of X and beta parameters vector)
  '''
      # Corrective measure: Double check appropriate dimensions of dataframes
  dim1 = X.shape[1]
  dim2 = B.shape[0]
  if  (dim1) == (dim2):
    predictions = X.dot(B.to_numpy())
    return (predictions)
  else:
    print('Cannot compute. Please check Dimensions!')
    print('Dimensions of design matrix Nxp: ', X.shape)
    print('Dimensions of Initialized Parameter Vector: (', len(B), 'x 1)')

# Output
yhat = total_predict(X, beta)
print('Total predictions, yhat:')
print(yhat)
print('')
print('yhat Transposed First few entries:')
print((yhat.T).head(3))

---

# RSS Function
def residual(Y, yhat):
  '''
  :param Y: true values
  :param yhat: predictions (i.e. dot product of X and parametric vector beta)
  :return: residual sum of sqaures
  '''
  dim1 = Y.shape
  dim2 = yhat.shape
  print("Y and yhat Dimensions Check:")
  print(dim1, 'and', dim2)
  rss = pd.DataFrame((Y.values - yhat.values)**2)
  return rss
```

```
# Output
residual_result = residual(Y,yhat)
print('rss:', residual_result)

---

# Cost Function
def cost(rss, tuning, b):
  '''
  :param b: parametric_vector_B
  :param rss: residual sum of squares
  :param tuning: tuning_parameter_vector
  :return: minimum cost computation
  '''
  # Checking pandas DataFrame dimensions
  tuning = pd.DataFrame(tuning)
  dim1, dim2, dim3 = rss.shape, tuning.shape, b.shape
  print('')
  print('rss Dimensions Check:', dim1)
  print('tuning parameter vector Dimensions Check:', dim2)
  print('b randomized vector Dimensions Check:', dim3)
  print('')
  # Cost Computation
  tobesummed = []
  for j in range(len(b)):
    compute = tuning @ ((b.iloc[j])**2)
    tobesummed.append(compute)
  regularization = sum(tobesummed)
  total_cost = rss.values + ((regularization).T).values
  # Convert to pandas Dataframe syntax
  return pd.DataFrame(total_cost)

# Output
rss1 = residual(Y, yhat)
cost_result = cost(rss1, tune, beta)
print('Cost Function Computation:')
print(cost_result)

---

# Gradient Descent Function
def gd(X,Y,tune,alpha,max_iter):
  '''
  :param X: our standardized Nxp design matrix, <class 'pandas.core.frame.DataFrame>
```

```
    :param Y: our N-dimensional centered y
    :param beta: our randomply initialized parametric vector B
    :param max_iter: max iterations alloted for convergence
    :param alpha: proof of when convergence occurs (i.e. 2*alpha*lambda <1)
    :param L: lambda value in tuning parameter vector
    :return beta_update, cost: total gradient calculation and cost computation

    This function is used for 'training' phase/step.
    '''
    # Computations
    b = pd.DataFrame(data=np.random.uniform(-1, 1, X.shape[1]))
    XB = pd.DataFrame((X.values).dot(b.values))
    Y_minus_XB = pd.DataFrame(Y.values - XB.values)
    X_T_dotprod_Y_minus_XB = (X.T).dot(Y_minus_XB)

    # Change to dataframe syntax for computations
    tuning = pd.DataFrame(tune)
    Lb = tuning.dot(b.T)
    # Making pretty dataframe for lambabeta - for faster computations
    d = pd.Series(tune)
    lambda_beta_df = (Lb.T).rename(columns = d, inplace = False)
    # We will use this dataframe to plot a graph later

    # Iterate for every column in lambda_beta dataframe and subtract XB column
    for i in lambda_beta_df.columns:
      compute = lambda_beta_df[i].values - (X_T_dotprod_Y_minus_XB.iloc[:,0].values)

    # Iterate for 1000 iterations - this yields convergence by given assumption
    for iteration in range(max_iter):
      b_temp = (2*alpha)*(pd.DataFrame(compute))
      b_update = b - b_temp
    return b_update, lambda_beta_df # return new updated beta vector

  # Output
  gd_computation = (gd(X, Y, tune, alpha, 1000)[0])
  to_plot = (gd(X, Y, tune, alpha, 1000)[1])
  print('For gradient descent:')
  print(gd_computation)
```

For the Cross Validation, listed below are the algorithms source codes. Functions listed are as follows:

1. Cross Validation Split Function: *cross_validation_split*
2. Cross Validation Function: *CV*

```
# Set K parameter for K-fold Cross Validation
K = 5

# Import Library for randomization
import random
from random import randrange

# Cross Validation Split Function
def cross_validation_split(dataset, folds= K):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / folds)
    for i in range(folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split

  ---
# Cross Validation Algorithm
def CV(X_d2, Y_d2, tune, alpha):
  '''
  Cross Validation Function
  :param X_d2: k-fold data dataframe
  :param Y_d2: true values fold data dataframe
  :param tune: tuning parameter of lambda values
  :param alpha: learning rate
  :return: beta updated dataframe for each fold, cost dataframe for each fold
  '''
  CV_5 = (gd(X_d2, Y_d2, tune, alpha, 1000))
  cv_beta = CV_5[0] # beta_update
  CV_5_beta = (cv_beta).rename(columns = groups, inplace = False)
  CV_5_cost = CV_5[1]
  CV_5_cost = (CV_5_cost.T).rename(columns = groups, inplace = False)
```

```
  return CV_5_beta, CV_5_cost

# Output for Viewing
# print('Cross Validation beta updates for each k-fold:')
# print(CV(X_d2, Y_d2, tune, alpha)[0])
# print('')
# print('Cost for each CV fold')
# print(CV(X_d2, Y_d2, tune, alpha)[1])

# Recommended Visual Output for Viewing for beta updates output:

CV_5 = (gd(X_d2, Y_d2, tune, alpha, 1000)[0]) # beta_update
CV_5_beta = (CV_5.T).rename(columns = groups, inplace = False)
CV_5_beta

# Recommended Visual Output for Viewing for K-folds cost output:

CV_5_cost = gd(X_d2, Y_d2, tune, alpha, 1000)[1].T # Cost
CV_5_cost= (CV_5_cost).rename(columns = groups, inplace = False)
CV_5_cost
```

You will need to initialize the folds as follows to use the Cross Validation functions mentioned above:

```
# Grabbing 5 data folds from X data
random.seed(1)                                # random seed
                                              # d2 = (training_data.iloc[:,: -1]).values
                                              # .values turnes into np array
d2 = X.values
X_folds = cross_validation_split(d2, K)       # cross validation
X_folds_df = (pd.DataFrame(X_folds))          # convert to dataframe
X_folds_df.T                                  # Transposed dataframe
                                              # Note: fold_size = total_rows / total_folds
groups = pd.Series(['k1', 'k2', 'k3', 'k4', 'k5'])
X_d2 = (X_folds_df.T).rename(columns = groups, inplace = False)
X_d2

---

# Grab the 5 datafolds for true value Y data
random.seed(1)                                   # random seed
# balance = (clean['Balance']).values          # np array
```

```
balance = Y.values
Y_folds = cross_validation_split(balance, K)   # cross validation
Y_folds_df = (pd.DataFrame(Y_folds))      # convert to dataframe
Y_folds_df.T                              # Transposed dataframe dim: 80 x 5
# Note: there are 80 rows per fold as we expected: fold_size = total_rows / K
Y_d2 = (Y_folds_df.T).rename(columns = groups, inplace = False)
Y_d2

---
```