Logistic Regression

YoungWoong Cho

July 2020

In this project, a binary classification dataset was chosen and logistic regression with stochastic gradient descent(SGA) as the optimization algorithm was implemented. SGD *without regularization* and SGD *with regularization* are implemented, and their performances were analyzed based on % correct on the test dataset. Lastly, likelihood functions with respect to iterations for unregularized and regularized logistic regression are plotted and compared.

Dataset : Wine Quality Classification

Source : https://www.kaggle.com/nareshbhat/wine-quality-binary-classification

Features : fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur, dioxide, total sulfur dioxide, density, pH, sulphates, alcohol

Label : quality

0.1 Preliminary works

1. Import libraries

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
```

2. Get a dataset

[2]:
```
from google.colab import files
data_to_load = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving wine.csv to wine.csv
```

[3]:
```
wine_data = pd.read_csv('wine.csv')
wine_data['quality'] = wine_data['quality'].replace('bad',0).replace('good',1)

wine_data.head()
```

```
[3]:    fixed acidity  volatile acidity  citric acid  ...  sulphates  alcohol
quality
0             7.4              0.70         0.00  ...       0.56      9.4
0
1             7.8              0.88         0.00  ...       0.68      9.8
0
2             7.8              0.76         0.04  ...       0.65      9.8
0
3            11.2              0.28         0.56  ...       0.58      9.8
1
4             7.4              0.70         0.00  ...       0.56      9.4
0

[5 rows x 12 columns]
```

3. Create train, test and validation sets

```
[ ]: X = np.concatenate( (np.ones((len(wine_data),1)), wine_data.iloc[:,0:-1].
      ↪to_numpy()), axis=1)
     y = wine_data.iloc[:,-1].to_numpy()

     train_X, test_X, train_Y, test_Y = train_test_split(X, y, test_size=0.2,␣
      ↪random_state=1)
     train_X, val_X, train_Y, val_Y = train_test_split(train_X, train_Y, test_size=0.
      ↪25, random_state=1)

     print("Data length: %d, Training data length: %d, Validation data length: %d,␣
      ↪Testing data length: %d" %(len(X), len(train_X), len(val_X), len(test_X)))
```

```
Data length: 1599, Training data length: 959, Validation data length: 320,
Testing data length: 320
```

4. Define basic functions for logistic regression

```
[ ]: # Sigmoid function
     # Probability of x being classified as 1
     def sig(x):
       return 1/(1+np.exp(-x))

     # Estimated classification using X and theta
     def est(X, theta):
       """
       Input:
         X: features from ith observation; p*1 vector where p: (# of features)
         theta: weighting coefficients; p*1 vector
       Output:
         Estimated classification of X
       """
```

```
  if theta.T.dot(X) >0.5:
    return 1
  else:
    return 0

# Likelihood
def L(X, Y, theta):
  """
  Input:
    X: features from ith observation; p*1 vector where p: (# of features)
    Y: label of ith observation; scalar
    theta: weighting coefficients; p*1 vector
  Output:
    Likelihood of weighted sum of X is classified as given label
  """
  h = sig(theta.T.dot(X))
  return h**Y * (1-h)**(1-Y)

# Log-Likelihood
def LL(X, Y, theta):
  """
  Input:
    X: features from ith observation; p*1 vector where p: (# of features)
    Y: label of ith observation; scalar
    theta: weighting coefficients; p*1 vector
  Output:
    Log-likelihood of weighted sum of X is classified as 1
  """
  h = sig(theta.T.dot(X))
  return Y*np.log(h) + (1-Y)*np.log(1 - h)
```

0.2 SGA without regularization

[ ]:
```
# Stochastic Gradient Ascent without regularization for ith observation
def SGA(X, Y, theta, alpha):
  """
  Input:
    X: features from ith observation; p*1 vector where p: (# of features)
    Y: label of ith observation; scalar
    theta: weighting coefficients; p*1 vector
    alpha: learning rate
  Output:
    updated theta; p*1 vector
  """
  h = sig(theta.T.dot(X))
  return theta + alpha*(Y - h)*X
```

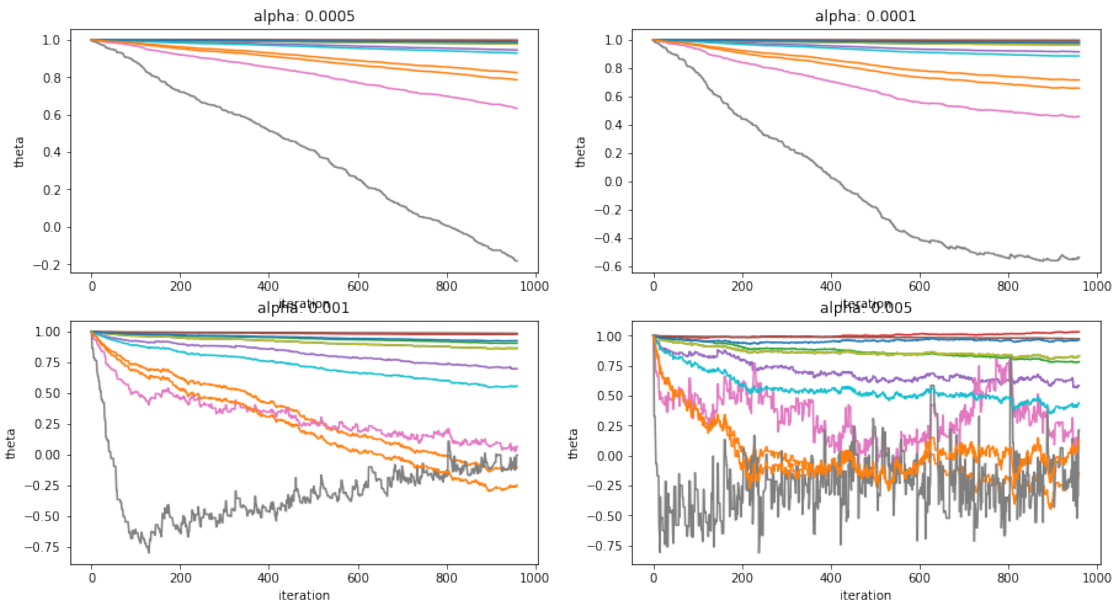
```python
def logistic_regression_theta(X, Y, alpha):
  """
  Input: alpha (step size)
  Output: p*n theta matrix; each column indicates theta values updated for␣
↪#(col_num) times
  """
  theta = np.ones((len(X[0]), 1)) #Initial guess of parameters
  # Iterate SGA for all observations
  # Each column will be a theta vector that is updated for #(col_num) times
  # Ex) theta = [ [theta updated 0 times].T [theta updated 1 times].T [theta␣
↪updated 2 times].T ]
  for i in range(len(X)):
    # Calculate the updated theta using SGA
    theta = np.concatenate((theta, SGA(X[i].T, Y[i], theta[:, -1], alpha).
↪reshape(len(X[0]), 1)), axis=1) # Compute SGA to calculate and store the␣
↪updated theta vector
  return theta

def logistic_regression_likelihood(X, Y, theta):
  """
  Input : p*n theta matrix; each column indicates theta values updated for␣
↪#(col_num) times
  Output: likelihood for each theta vector
  """
  # Likelihood for each updated theta
  likelihood = []

  for i in range(len(theta[0])): # Iterate through each theta
    likelihood_temp = 1 # Reset the value to 0 for i-th theta
    for j in range(len(X)): # Iterate through each dataset
      likelihood_temp = likelihood_temp * L(X[j].T, Y[j], theta[:, i]) #␣
↪Multiply likelihood for this theta value for entire training set
    likelihood.append(likelihood_temp) # Append likelihood for this theta
  return likelihood

def logistic_regression_log_likelihood(X, Y, theta):
  """
  Input : p*n theta matrix; each column indicates theta values updated for␣
↪#(col_num) times
  Output: likelihood for each theta vector
  """
  # Log likelihood for each updated theta
  log_likelihood = []

  for i in range(len(theta[0])): # Iterate through each theta
```

```
    log_likelihood_temp = 0 # Reset the value to 0 for i-th theta
    for j in range(len(X)): # Iterate through each dataset
      log_likelihood_temp = log_likelihood_temp + LL(X[j].T, Y[j], theta[:, i])␣
↪# Add log-likelihood to this theta value for entire training set
    log_likelihood.append(log_likelihood_temp) # Append log-likelihood for this␣
↪theta
  return log_likelihood
```

Each thea value as a function of the number of iteration through the dataset for a given step size is plotted to qualitatively analyze the optimal step size. If we have too small a step size, thetas would not reach the equilibrium point. On the other hand, if we make a step size too large, thetas would not converge and oscillate around the equilibrium point.

[ ]:
```
# Plot each theta as SGA is iterated
def plot_theta(X, Y, alpha):
  """
  Input: alpha(step size)
  Output: Plot n-th component of theta
  """
  theta = logistic_regression_theta(X, Y, alpha) # p*N theta matrix
  for i in range(len(theta)):
    plt.plot(theta[i])

  plt.xlabel('iteration')
  plt.ylabel('theta')
  #plt.legend(['theta1', 'theta2', 'theta3', 'theta4', 'theta5', 'theta6',␣
↪'theta7', 'theta8', 'theta9', 'theta10'], loc=3)
```

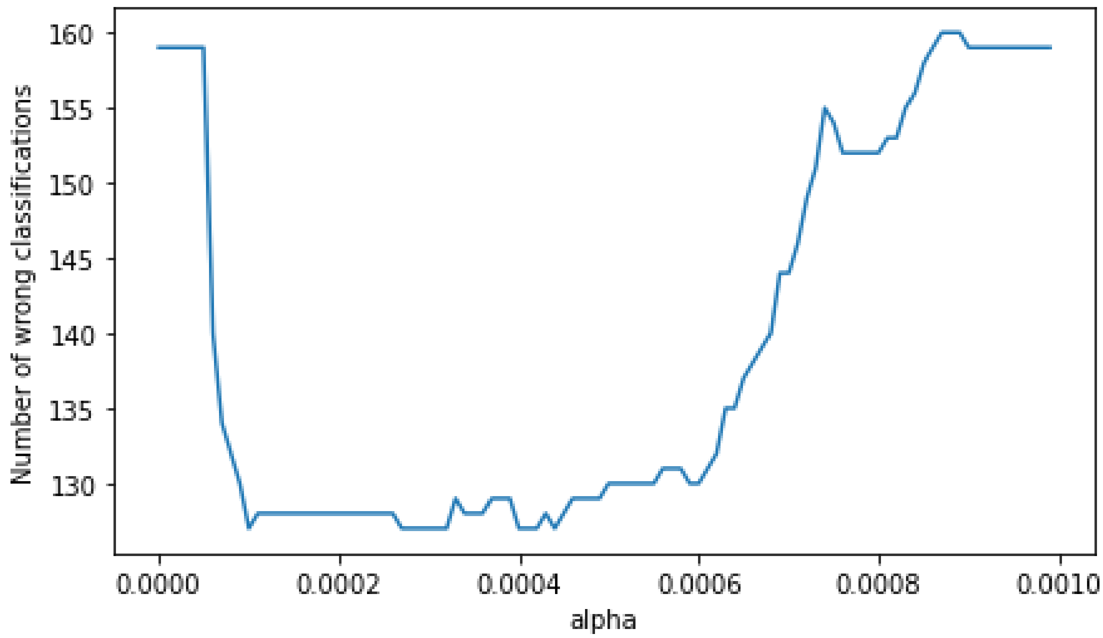
[ ]:
```
plt.rcParams["figure.figsize"] = (15,8)

plt.subplot(2, 2, 1)
plt.title("alpha: 0.0005")
plot_theta(train_X, train_Y, 0.00005)

plt.subplot(2, 2, 2)
plt.title("alpha: 0.0001")
plot_theta(train_X, train_Y, 0.0001)

plt.subplot(2, 2, 3)
plt.title("alpha: 0.001")
plot_theta(train_X, train_Y, 0.001)

plt.subplot(2, 2, 4)
plt.title("alpha: 0.005")
plot_theta(train_X, train_Y, 0.005)
plt.show()
```



From the plots above, it can be deduced that step size of 0.0005 is too small and 0.001 is too large. One can expect the optimal step size to be around 0.0001. The optimal alpha was numerically calculated by plotting the log-likelihood with different step sizes.

[ ]:
```
# Find optimal alpha
alphas = np.arange(0, 0.001, 0.00001)
wrongs = []

for a in alphas:
  num_wrong = 0

  theta = logistic_regression_theta(train_X, train_Y, a)

  for i in range(len(val_X)):
    X = val_X[i].reshape(len(val_X[0]), 1)
    Y = val_Y[i]
    if (Y!=est(X, theta[:, -1])):
      num_wrong = num_wrong + 1

  wrongs.append(num_wrong)
  #print("Alpha: %.4f\nTotal number of data: %d\nNumber of wrong classifications:
 ↪ %d\nPercent error: %.2f%%\n" %(a, len(test_X), num_wrong, num_wrong/
 ↪len(test_X)*100))

plt.rcParams["figure.figsize"] = (7,4)
plt.xlabel('alpha')
plt.ylabel('Number of wrong classifications')
```

```
plt.plot(alphas, wrongs)
plt.show()

print("Optimal alpha: %.4f" %alphas[wrongs.index(min(wrongs))])
```




```
Optimal alpha: 0.0001
```

As expected, optimal step size occurs at 0.0001.

[ ]:
```
# Test
theta = logistic_regression_theta(train_X, train_Y,  alphas[wrongs.
 ↪index(min(wrongs))])

num_wrong = 0
for i in range(len(test_X)):
  X = test_X[i].reshape(len(test_X[0]), 1)
  Y = test_Y[i]
  if (Y!=est(X, theta[:,-1])):
    num_wrong = num_wrong + 1

print("Optimal alpha: %.4f\nTotal number of data: %d\nNumber of wrong␣
 ↪classifications: %d\nPercent error: %.2f%%" %(alphas[wrongs.
 ↪index(min(wrongs))], len(test_X), num_wrong, num_wrong/len(test_X)*100))
```

```
Optimal alpha: 0.0001
Total number of data: 320
Number of wrong classifications: 116
```

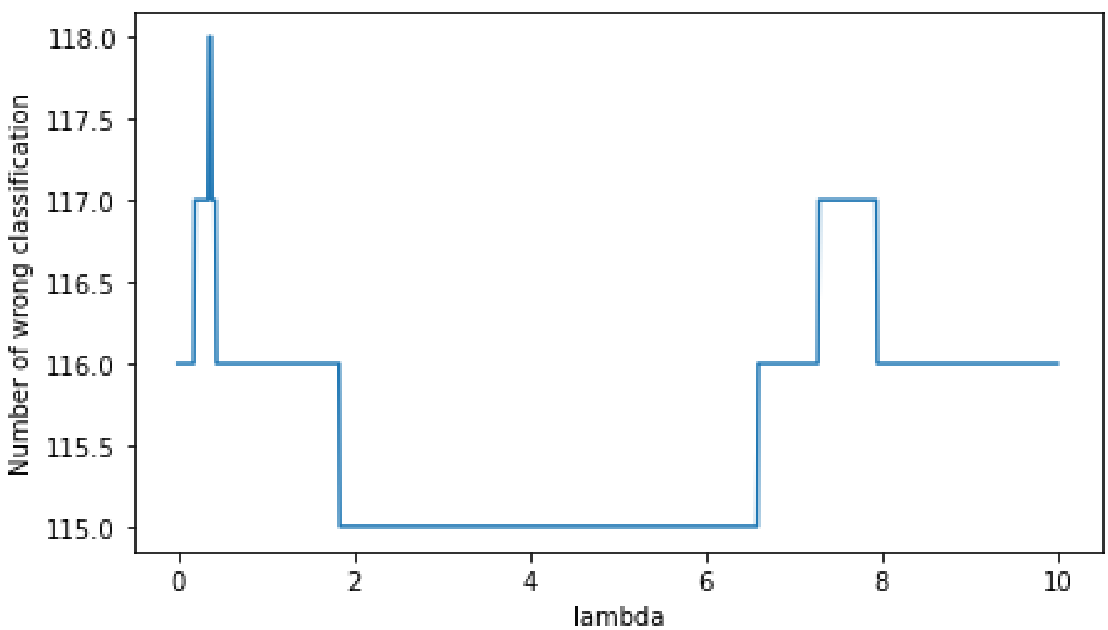
```
Percent error: 36.25%
```

0.3 SGA with regularization

[ ]:
```python
# Stochastic Gradient Ascent with L2 regularization
def SGA_L2(X, Y, theta, alpha, lamb):
  """
  Input:
    X: features from ith observation; p*1 vector where p: (# of features)
    Y: label of ith observation; scalar
    theta: weighting coefficients; p*1 vector
    alpha: learning rate
  Output:
    updated theta; p*1 vector
  """
  h = sig(theta.T.dot(X))
  add_term = alpha*(((Y - h)*X) - 2*lamb*theta)
  return theta + add_term

def logistic_regression_theta_reg(X, Y, alpha, lamb):
  """
  Input:
    alpha (step size)
    lambda (tuning param)
  Output:
    p*n theta matrix; each column indicates theta values updated for #(col_num)␣
 ↪times
  """
  theta = np.ones((len(X[0]), 1)) #Initial guess of parameters
  # Iterate SGA for all observations
  # Each column will be a theta vector that is updated for #(col_num) times
  # Ex) theta = [ [theta updated 0 times].T [theta updated 1 times].T [theta␣
 ↪updated 2 times].T ]
  for i in range(len(X)):
    # Calculate the updated theta using SGA
    theta = np.concatenate((theta, SGA_L2(X[i].T, Y[i], theta[:, -1], alpha,␣
 ↪lamb).reshape(len(X[0]), 1)), axis=1) # Compute SGA_L2 to calculate and store␣
 ↪the updated theta vector
  return theta
```

[ ]:
```python
# Find best tuning parameter using validation dataset
val_wrongs = []
lambdas = np.linspace(0, 10, 1000)

for lamb in lambdas:
  num_wrong = 0
```

```
  theta = logistic_regression_theta_reg(train_X, train_Y, alphas[wrongs.
↪index(min(wrongs))], lamb)

  for i in range(len(test_X)):
    X = test_X[i].reshape(len(test_X[0]), 1)
    Y = test_Y[i]
    if (Y!=est(X, theta[:, -1])):
      num_wrong = num_wrong + 1

  val_wrongs.append(num_wrong)

plt.xlabel('lambda')
plt.ylabel('Number of wrong classification')
plt.plot(lambdas, val_wrongs)
plt.show()

print("Using optimal step size %.4f, optimal lambda: %.4f" %(alphas[wrongs.
↪index(min(wrongs))], lambdas[val_wrongs.index(min(val_wrongs))]))
```



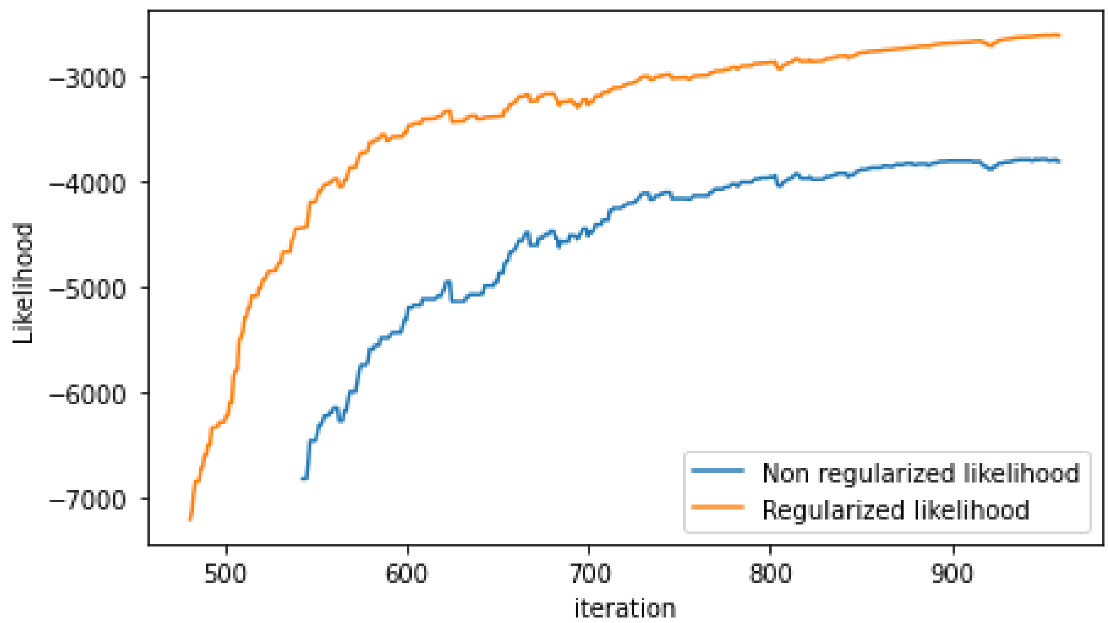

```
Using optimal step size 0.0001, optimal lambda: 1.8418
```

[ ]:
```
#Test
theta = logistic_regression_theta_reg(train_X, train_Y, alphas[wrongs.
↪index(min(wrongs))], lambdas[val_wrongs.index(min(val_wrongs))])

num_wrong = 0
```

```python
for i in range(len(test_X)):
  X = test_X[i].reshape(len(test_X[0]), 1)
  Y = test_Y[i]
  if (Y!=est(X, theta[:, -1])):
    num_wrong = num_wrong + 1

print("Optimal setp size: %.4f\nOptimal Lambda: %.4f\nTotal number of data:␣
 ↪%d\nNumber of wrong classifications: %d\nPercent error: %.2f%%"␣
 ↪%(alphas[wrongs.index(min(wrongs))], lambdas[val_wrongs.
 ↪index(min(val_wrongs))], len(test_X), num_wrong, num_wrong/len(test_X)*100))
```

```
Optimal setp size: 0.0001
Optimal Lambda: 1.8418
Total number of data: 320
Number of wrong classifications: 115
Percent error: 35.94%
```

[ ]:
```python
non_reg_theta = logistic_regression_theta(train_X, train_Y,  alphas[wrongs.
 ↪index(min(wrongs))])
non_reg_LL = logistic_regression_log_likelihood(train_X, train_Y, non_reg_theta)

reg_theta = logistic_regression_theta_reg(train_X, train_Y, alphas[wrongs.
 ↪index(min(wrongs))], lambdas[val_wrongs.index(min(val_wrongs))])
reg_LL = logistic_regression_log_likelihood(train_X, train_Y, reg_theta)

plt.rcParams["figure.figsize"] = (7,4)
plt.xlabel('iteration')
plt.ylabel('Log-likelihood')
plt.plot(non_reg_LL)
plt.plot(reg_LL)
plt
plt.legend(['Non regularized likelihood', 'Regularized likelihood'])
plt.show()
```


Likelihood
−3000
−4000
−5000
−6000
−7000
500
600
700
800
900
iteration
Non regularized likelihood
Regularized likelihood