HM1: Logistic Regression.

Name: Harris Spahic

For this assignment, you will build 6 models. You need to train Logistic Regression/Regularized Logistic Regression each with Batch Gradient Descent, Stochastic Gradient Descent and Mini Batch Gradient Descent. Also, you should plot their objective values versus epochs and compare their training and testing accuracy. You will need to tune the parameters a little bit to obtain reasonable results.

You do not have to follow the following procedure. You may implement your own functions and methods, but you need to show your results and plots.

```
# Load Packages
import numpy
import pandas as pd
from sklearn.model_selection import train_test_split

import random
```

1. Data processing

- Download the Breast Cancer dataset from canvas or from https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(diagnostic)
- Load the data.
- Preprocess the data.

1.1. Load the data

```
data = pd.read_csv("data-1.csv")
```

1.2 Examine and clean data

```
# Some columns may not be useful for the model (For example, the first column contains ID number which may be irrelavant).
# You need to get rid of the ID number feature.
# Also you should transform target labels in the second column from 'B' and 'M' to 1 and -1.
data.head()
data = data.drop(columns = ["id", "Unnamed: 32"])

def alter_diagnosis(row):
  if(row.diagnosis == 'M'):
    row.diagnosis = -1
  else:
    row.diagnosis = 1
  return row

data = data.apply(alter_diagnosis, axis="columns")
```

```
print(data)
```

```
     565         -1        20.13         28.25          131.20     1261.0
     566         -1        16.60         28.08          108.30      858.1
     567         -1        20.60         29.33          140.10     1265.0
     568          1         7.76         24.54           47.92      181.0
```

```
565          0.09780           0.10340          0.14400              0.09791
566          0.08455           0.10230          0.09251              0.05302
567          0.11780           0.27700          0.35140              0.15200
568          0.05263           0.04362          0.00000              0.00000

     symmetry_mean  ...  radius_worst  texture_worst  perimeter_worst  \
0           0.2419  ...        25.380          17.33           184.60
1           0.1812  ...        24.990          23.41           158.80
2           0.2069  ...        23.570          25.53           152.50
3           0.2597  ...        14.910          26.50            98.87
4           0.1809  ...        22.540          16.67           152.20
..             ...  ...           ...            ...              ...
564         0.1726  ...        25.450          26.40           166.10
565         0.1752  ...        23.690          38.25           155.00
566         0.1590  ...        18.980          34.12           126.70
567         0.2397  ...        25.740          39.42           184.60
568         0.1587  ...         9.456          30.37            59.16

     area_worst  smoothness_worst  compactness_worst  concavity_worst  \
0        2019.0           0.16220            0.66560           0.7119
1        1956.0           0.12380            0.18660           0.2416
2        1709.0           0.14440            0.42450           0.4504
3         567.7           0.20980            0.86630           0.6869
4        1575.0           0.13740            0.20500           0.4000
..          ...               ...                ...              ...
564      2027.0           0.14100            0.21130           0.4107
565      1731.0           0.11660            0.19220           0.3215
566      1124.0           0.11390            0.30940           0.3403
567      1821.0           0.16500            0.86810           0.9387
568       268.6           0.08996            0.06444           0.0000

     concave points_worst  symmetry_worst  fractal_dimension_worst
0                  0.2654          0.4601                  0.11890
1                  0.1860          0.2750                  0.08902
2                  0.2430          0.3613                  0.08758
3                  0.2575          0.6638                  0.17300
4                  0.1625          0.2364                  0.07678
..                    ...             ...                      ...
564                0.2216          0.2060                  0.07115
565                0.1628          0.2572                  0.06637
566                0.1418          0.2218                  0.07820
567                0.2650          0.4087                  0.12400
568                0.0000          0.2871                  0.07039

[569 rows x 31 columns]
```

1.3. Partition to training and testing sets

```
# You can partition using 80% training data and 20% testing data. It is a commonly used ratio in machine learning.
features = list(data.columns.values)
features.remove("diagnosis")
X = data.loc[:, features]
y = data.loc[:, ["diagnosis"]]
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

1.4. Feature scaling

Use the standardization to transform both training and test features

```
# Standardization
import numpy

# calculate mu and sig using the training set
d = x_train.shape[1]
mu = numpy.mean(x_train, axis=0)
sig = numpy.std(x_train, axis=0)

# transform the training features
x_train = (x_train - mu) / (sig + 1E-6)

# transform the test features
x_test = (x_test - mu) / (sig + 1E-6)

print('test mean = ')
print(numpy.mean(x_test, axis=0))
```

```
print('test std = ')
print(numpy.std(x_test, axis=0))

x_train, y_train, x_test, y_test = x_train.to_numpy(), y_train.to_numpy(), x_test.to_numpy(), y_test.to_numpy()
```

```
compactness_mean          -0.161125
concavity_mean            -0.173081
concave points_mean       -0.118854
symmetry_mean             -0.164165
fractal_dimension_mean    -0.111343
radius_se                 -0.041021
texture_se                -0.056817
perimeter_se              -0.083738
area_se                   -0.037372
smoothness_se              0.101092
compactness_se            -0.090339
concavity_se              -0.077154
concave points_se          0.026614
symmetry_se               -0.067501
fractal_dimension_se      -0.049114
radius_worst              -0.016442
texture_worst             -0.056242
perimeter_worst           -0.045007
area_worst                -0.030742
smoothness_worst          -0.089337
compactness_worst         -0.183273
concavity_worst           -0.172896
concave points_worst      -0.097847
symmetry_worst            -0.155435
fractal_dimension_worst   -0.104520
dtype: float64
test std = 
radius_mean                0.878722
texture_mean               0.844264
perimeter_mean             0.865596
area_mean                  0.833381
smoothness_mean            0.938096
compactness_mean           0.740519
concavity_mean             0.732034
concave points_mean        0.800389
symmetry_mean              0.863104
fractal_dimension_mean     0.908649
radius_se                  0.809816
texture_se                 0.797983
perimeter_se               0.742678
area_se                    0.715284
smoothness_se              1.281863
compactness_se             0.923997
concavity_se               0.741775
concave points_se          0.972605
symmetry_se                0.783677
fractal_dimension_se       0.869523
radius_worst               0.894132
texture_worst              0.812853
perimeter_worst            0.861239
area_worst                 0.861553
smoothness_worst           0.906860
compactness_worst          0.754433
concavity_worst            0.784418
concave points_worst       0.822799
symmetry_worst             0.807468
fractal_dimension_worst    0.942904
dtype: float64
```

2. Logistic Regression Model

The objective function is $Q(w; X, y) = \frac{1}{n} \sum_{i=1}^{n} \log \Big(1 + \exp \big(-y_i x_i^T w\big)\Big) + \frac{\lambda}{2} \|w\|_2^2$.

When $\lambda = 0$, the model is a regular logistic regression and when $\lambda > 0$, it essentially becomes a regularized logistic regression.

```
# Calculate the objective function value, or loss
# Inputs:
#     w: weight: d-by-1 matrix
#     x: data: n-by-d matrix
#     y: label: n-by-1 matrix
#     lam: regularization parameter: scalar
# Return:
#     objective function value, or loss (scalar)
```

```
def objective(w, x, y, lam):
    assert(x.shape[0] == y.shape[0])
    assert(x.shape[1] == w.shape[0])

    sum = 0; n = x.shape[0]
    for i in range(n):
      z = numpy.matmul(numpy.multiply(y[i,:], x[i, :]), w)
      result = numpy.log(1 + numpy.exp(-z))
      sum += result

    bias = (lam / 2) * numpy.linalg.norm(w)
    return sum / n + bias
```

3. Numerical optimization

3.1. Gradient descent

The gradient at w for regularized logistic regression is $g = -\frac{1}{n}\sum_{i=1}^{n}\frac{y_i x_i}{1+\exp(y_i x_i^T w)} + \lambda w$

```
# Calculate the gradient
# Inputs:
#     w: weight: d-by-1 matrix
#     x: data: n-by-d matrix
#     y: label: n-by-1 matrix
#     lam: regularization parameter: scalar
# Return:
#     g: gradient: d-by-1 matrix

def gradient(w, x, y, lam):
    assert(x.shape[0] == y.shape[0])
    assert(x.shape[1] == w.shape[0])

    sum = 0; n = x.shape[0]
    for i in range(n):
      z = numpy.matmul(numpy.multiply(y[i], x[i]), w)
      numerator = numpy.multiply(y[i], x[i])
      sum -= numerator / (1 + numpy.exp(z))

    sum = sum/n
    return sum + lam * w

test_weights = numpy.ones(x_train.shape[1])
print(gradient(test_weights, x_train, y_train, 0.5))
```

```
[1.13940677 0.87052781 1.15847    1.12395891 0.90443253 1.14821285
 1.21194901 1.25241669 0.88872401 0.64285182 1.01836943 0.53928533
 1.02235917 0.99033007 0.49285693 0.91750294 0.84703261 0.98954383
 0.57139691 0.71992744 1.18492835 0.91531349 1.1996699  1.15143909
 0.93854961 1.12378615 1.1748716  1.26587501 0.93253374 0.91188311]
```

```
# Gradient descent for solving logistic regression
# You will need to do iterative processes (loops) to obtain optimal weights in this function

# Inputs:
#     x: data: n-by-d matrix
#     y: label: n-by-1 matrix
#     lam: scalar, the regularization parameter
#     learning_rate: scalar
#     w: weights: d-by-1 matrix, initialization of w
#     max_epoch: integer, the maximal epochs
# Return:
#     w: weights: d-by-1 matrix, the solution
#     objvals: a record of each epoch's objective value

def gradient_descent(x, y, lam, learning_rate, w, max_epoch=100):
    objvals = []

    for epoch in range(max_epoch):
      obj = objective(w, x, y, lam)
      w -= gradient(w, x, y, lam) * learning_rate
```

```
        objvals.append(obj)

    return w, objvals
```

Use gradient_descent function to obtain your optimal weights and a list of objective values over each epoch.

```
# Train logistic regression
# You should get the optimal weights and a list of objective values by using gradient_descent function.

lgr_weights = numpy.ones(x_train.shape[1])
lgr_weights, lgr_objvals = gradient_descent(x_train, y_train, 0, 0.1, lgr_weights, max_epoch = 100)
print(lgr_objvals)
```

```
962955902491, 0.12657456898587133, 0.12586834556813062, 0.125177143374971, 0.1245005003807674, 0.12383797337399735, 0.12318913691930937]
```

```
# Train regularized logistic regression
# You should get the optimal weights and a list of objective values by using gradient_descent function.

r_lgr_weights = numpy.ones(x_train.shape[1])
r_lgr_weights, r_lgr_objvals = gradient_descent(x_train, y_train, 0.5, 0.1, r_lgr_weights, max_epoch = 100)
print(r_lgr_objvals)
```

```
584614440884, 0.4073563980804533, 0.4073545052021609, 0.40735276803964615, 0.40735117317451647, 0.40734970840815515, 0.4073483626439727]
```

3.2. Stochastic gradient descent (SGD)

Define new objective function $Q_i(w) = \log\Big(1 + \exp\big(-y_i x_i^T w\big)\Big) + \frac{\lambda}{2}\|w\|_2^2$.

The stochastic gradient at w is $g_i = \frac{\partial Q_i}{\partial w} = -\frac{y_i x_i}{1+\exp(y_i x_i^T w)} + \lambda w$.

You may need to implement a new function to calculate the new objective function and gradients.

```
# Calculate the objective Q_i and the gradient of Q_i
# Inputs:
#     w: weights: d-by-1 matrix
#     xi: data: 1-by-d matrix
#     yi: label: scalar
#     lam: scalar, the regularization parameter
# Return:
#     obj: scalar, the objective Q_i
#     g: d-by-1 matrix, gradient of Q_i

def stochastic_objective_gradient(w, xi, yi, lam):
    assert(w.shape[0] == xi.shape[0])
    z = numpy.matmul(numpy.multiply(yi, xi), w)
    obj = numpy.log(1 + numpy.exp(-z)) + (lam/2) * numpy.linalg.norm(w)
    g = -(numpy.multiply(yi, xi) / (1 + numpy.exp(z))) + lam * w

    return obj, g

# weights = numpy.ones(x_train.shape[1])
# sum = 0
# for i in range(x_train.shape[0]):
#   sum += stochastic_objective_gradient(weights, x_train[i], y_train[i], 0)[0]

# print(sum / x_train.shape[0])
```

Hints:

1. In every epoch, randomly permute the n samples.
2. Each epoch has n iterations. In every iteration, use 1 sample, and compute the gradient and objective using the `stochastic_objective_gradient` function. In the next iteration, use the next sample, and so on.

```
# SGD for solving logistic regression
# You will need to do iterative process (loops) to obtain optimal weights in this function
```

```
# Inputs:
#     x: data: n-by-d matrix
#     y: label: n-by-1 matrix
#     lam: scalar, the regularization parameter
#     learning_rate: scalar
#     w: weights: d-by-1 matrix, initialization of w
#     max_epoch: integer, the maximal epochs
# Return:
#
#     w: weights: d-by-1 matrix, the solution
#     objvals: a record of each epoch's objective value
#     Record one objective value per epoch (not per iteration)


def sgd(x, y, lam, learning_rate, w, max_epoch=100):
    objvals = []
    n = x.shape[0]

    for epoch in range(max_epoch):
      obj_sum = 0
      sample_index = numpy.random.permutation(n)
      for i in range(n):
        iter_obj, gradient = stochastic_objective_gradient(w, x[sample_index[i]], y[sample_index[i]], lam)
        w -= learning_rate * gradient
        obj_sum += iter_obj
      objvals.append(obj_sum /n)

    return w, objvals
```

Use sgd function to obtain your optimal weights and a list of objective values over each epoch.

```
# Train logistic regression
# You should get the optimal weights and a list of objective values by using gradient_descent function.

slg_weights = numpy.ones(x_train.shape[1])
slg_weights, slg_obj = sgd(x_train, y_train, 0, 0.001, slg_weights, max_epoch=100)
print(slg_obj)
```

```
518604437675, 0.0730436917086196, 0.0728395374211481, 0.07263813208837284, 0.07243995124277748, 0.0722444984201531, 0.07205245337860205]
```

```
# Train regularized logistic regression
# You should get the optimal weights and a list of objective values by using gradient_descent function.

slgr_weights = numpy.ones(x_train.shape[1])
slgr_weights, slgr_obj = sgd(x_train, y_train, 0.5, 0.001, slgr_weights, max_epoch=100)
print(slgr_obj)
```

```
559272683925, 0.4078921016951469, 0.4077888513741375, 0.40786570444193165, 0.40777192627166364, 0.4079180355522985, 0.40785379867751903]
```

3.3 Mini-Batch Gradient Descent (MBGD)

Define $Q_I(w) = \frac{1}{b} \sum_{i \in I} \log \Big(1 + \exp \big(- y_i x_i^T w \big) \Big) + \frac{\lambda}{2} \|w\|_2^2$, where I is a set containing b indices randomly drawn from $\{1, \cdots, n\}$ without replacement.

The stochastic gradient at w is $g_I = \frac{\partial Q_I}{\partial w} = \frac{1}{b} \sum_{i \in I} \frac{-y_i x_i}{1 + \exp(y_i x_i^T w)} + \lambda w$.

You may need to implement a new function to calculate the new objective function and gradients.

```
# Calculate the objective Q_I and the gradient of Q_I
# Inputs:
#     w: weights: d-by-1 matrix
#     xi: data: b-by-d matrix
#     yi: label: scalar
#     lam: scalar, the regularization parameter
# Return:
#     obj: scalar, the objective Q_i
#     g: d-by-1 matrix, gradient of Q_i
```

```
def mb_objective_gradient(w, xi, yi, lam):
    assert(w.shape[0] == xi.shape[1])

    obj_sum = 0
    grad_sum = 0
    b = xi.shape[0]

    for i in range(b):
      z = numpy.matmul(numpy.multiply(yi[i], xi[i]), w)
      obj_sum += numpy.log(1 + numpy.exp(-z))
      grad_sum = grad_sum - numpy.multiply(yi[i], xi[i]) / (1 + numpy.exp(z))

    return (obj_sum + (lam / 2) * numpy.linalg.norm(w))/b, (grad_sum + lam * w)/b
```

Hints:

1. In every epoch, randomly permute the n samples (just like SGD).
2. Each epoch has $\frac{n}{b}$ iterations. In every iteration, use b samples, and compute the gradient and objective using the `mb_objective_gradient` function. In the next iteration, use the next b samples, and so on.

```
# MBGD for solving logistic regression
# You will need to do iterative process (loops) to obtain optimal weights in this function

# Inputs:
#     x: data: n-by-d matrix
#     y: label: n-by-1 matrix
#     lam: scalar, the regularization parameter
#     learning_rate: scalar
#     w: weights: d-by-1 matrix, initialization of w
#     max_epoch: integer, the maximal epochs
# Return:
#     w: weights: d-by-1 matrix, the solution
#     objvals: a record of each epoch's objective value
#     Record one objective value per epoch (not per iteration)

def mbgd(x, y, lam, learning_rate, w, max_epoch=100):
    b = 20
    obj_vals = []
    obj_sum = 0

    for epoch in range(max_epoch):
      obj_sum = 0

      for i in range(x.shape[0] // b):
        selected_i = numpy.random.choice(x.shape[0], b)
        xi, yi = x[selected_i, :], y[selected_i, :]
        obj, grad = mb_objective_gradient(w, xi, yi, lam)

        w -= learning_rate * grad
        obj_sum += obj

      obj_sum = obj_sum / (x.shape[0] // b)

      obj_vals.append(obj_sum)


    return w, obj_vals
```

Use mbgd function to obtain your optimal weights and a list of objective values over each epoch.

```
# Train logistic regression
# You should get the optimal weights and a list of objective values by using gradient_descent function.

mb_weights = numpy.ones(x_train.shape[1])
mb_weights, mb_obj = mbgd(x_train, y_train, 0, 0.02, mb_weights, max_epoch = 100)
print(mb_obj)
```

```
548138134, 0.07923002252209797, 0.07277606875378402, 0.07423048392036806, 0.08093883005091738, 0.08837279802337598, 0.08419970576001946]
```

```
# Train regularized logistic regression
# You should get the optimal weights and a list of objective values by using gradient_descent function.
```

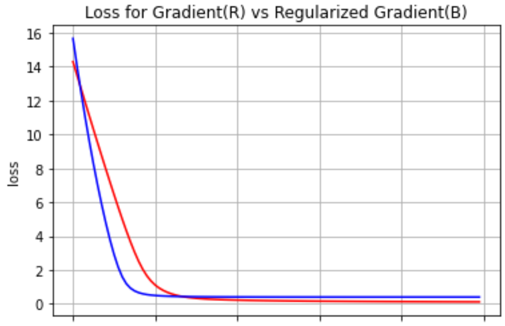
```
rmb_weights = numpy.ones(x_train.shape[1])
rmb_weights, rmb_obj = mbgd(x_train, y_train, 0.5, 0.02, rmb_weights, max_epoch = 100)
print(rmb_obj)
```

```
028582735888, 0.11770280500565562, 0.11404101395127282, 0.1036164597220778, 0.11550419776097237, 0.13488332668542524, 0.125694198373774]
```

4. Compare GD, SGD, MBGD

Plot objective function values against epochs.

```
import matplotlib.pyplot as plt
%matplotlib inline

lgr_objvals, r_lgr_objvals, slg_obj, slgr_obj, mb_obj, rmb_obj;
epochs = numpy.arange(100)

fig, ax = plt.subplots()
ax.plot(epochs, lgr_objvals, color = "red")
ax.plot(epochs, r_lgr_objvals, color = "blue")
ax.set(xlabel = "epoch #", ylabel = "loss", title = "Loss for Gradient(R) vs Regularized Gradient(B)")
ax.grid()

plt.show()

fig, ax = plt.subplots()
ax.plot(epochs, slg_obj, color = "red")
ax.plot(epochs, slgr_obj, color = "blue")
ax.set(xlabel = "epoch #", ylabel = "loss", title = "Loss for Gradient(R) vs Regularized Gradient(B)")
ax.grid()

plt.show()

fig, ax = plt.subplots()
ax.plot(epochs, mb_obj, color = "red")
ax.plot(epochs, rmb_obj, color = "blue")
ax.set(xlabel = "epoch #", ylabel = "loss", title = "Loss for Gradient(R) vs Regularized Gradient(B)")
ax.grid()

plt.show()
```



5. Prediction

Compare the training and testing accuracy for logistic regression and regularized logistic regression.

```
# Predict class label
# Inputs:
#     w: weights: d-by-1 matrix
#     X: data: m-by-d matrix
# Return:
#     f: m-by-1 matrix, the predictions
from decimal import Decimal

def predict(w, X):
    y_pred = numpy.matmul(X, w)
    fix_pred = lambda x: 1 if x > 0 else -1
    y_pred = [fix_pred(x) for x in y_pred]

    return y_pred

def calc_accuracy(y_pred, y_train):
    y_train = numpy.squeeze(y_train)
    return Decimal(1 - numpy.count_nonzero(y_train - y_pred) / len(y_pred))
```

```
# evaluate training error of logistic regression and regularized version

print(calc_accuracy(predict(lgr_weights, x_train), y_train))
print(calc_accuracy(predict(r_lgr_weights, x_train), y_train))

print(calc_accuracy(predict(slg_weights, x_train), y_train))
print(calc_accuracy(predict(slgr_weights, x_train), y_train))

print(calc_accuracy(predict(mb_weights, x_train), y_train))
print(calc_accuracy(predict(rmb_weights, x_train), y_train))
```

```
0.9516483516483515980866059180698357522487640380859375
0.95824175824175827909101599288987927138805389404296875
0.98681318681318686003578477539122104644775390625
0.95824175824175827909101599288987927138805389404296875
0.98021978021978017903137470057117752730846405029296875
0.984615384615384670041748904623091220855712890625
```

```
# evaluate testing error of logistic regression and regularized version

print(calc_accuracy(predict(lgr_weights, x_test), y_test))
print(calc_accuracy(predict(r_lgr_weights, x_test), y_test))

print(calc_accuracy(predict(slg_weights, x_test), y_test))
print(calc_accuracy(predict(slgr_weights, x_test), y_test))

print(calc_accuracy(predict(mb_weights, x_test), y_test))
print(calc_accuracy(predict(rmb_weights, x_test), y_test))
```

```
0.97368421052631581869007959539885632693767547607421875
0.9912280701754385692225923776277340948581695556640625
0.982456140350877138445184755255468189716339111328125
0.9912280701754385692225923776277340948581695556640625
0.982456140350877138445184755255468189716339111328125
0.9912280701754385692225923776277340948581695556640625
```

6. Parameters tuning

In this section, you may try different combinations of parameters (regularization value, learning rate, etc) to see their effects on the model. (Open ended question)

```
# Retrain each model on lower epoch size & larger regularization term

# Normal Gradient Descent
lgr_weights = numpy.ones(x_train.shape[1])
lgr_weights, lgr_objvals = gradient_descent(x_train, y_train, 0, 0.1, lgr_weights, max_epoch = 50)
print(lgr_objvals)

# Regularized Gradient Descent
r_lgr_weights = numpy.ones(x_train.shape[1])
r_lgr_weights, r_lgr_objvals = gradient_descent(x_train, y_train, 1, 0.1, r_lgr_weights, max_epoch = 50)
print(r_lgr_objvals)

# Stochastic Gradient Descent
slg_weights = numpy.ones(x_train.shape[1])
slg_weights, slg_obj = sgd(x_train, y_train, 0, 0.001, slg_weights, max_epoch=50)
print(slg_obj)

# Regularized Stochastic Gradient Descent
slgr_weights = numpy.ones(x_train.shape[1])
slgr_weights, slgr_obj = sgd(x_train, y_train, 1, 0.001, slgr_weights, max_epoch=50)
print(slgr_obj)

# Mini-batch Gradient Descent
mb_weights = numpy.ones(x_train.shape[1])
mb_weights, mb_obj = mbgd(x_train, y_train, 0, 0.02, mb_weights, max_epoch = 50)
print(mb_obj)

# Regularized Mini-batch Gradient Descent
rmb_weights = numpy.ones(x_train.shape[1])
rmb_weights, rmb_obj = mbgd(x_train, y_train, 1, 0.02, rmb_weights, max_epoch = 50)
print(rmb_obj)
```

```
, 0.2041676038977612, 0.20046554657772966, 0.19699350859680445, 0.1937270609582381, 0.1906452966986202]
652754631, 0.7034986530890106, 0.703498721065035, 0.7034987736311373, 0.7034988142432816]
420115769, 0.09156846843045098, 0.09087682520581093, 0.09020961166980647, 0.08956719345446652, 0.08894738778124427, 0.08835110854661737]
43114656165447, 0.7045669214000488, 0.7048961069593797, 0.704804655745422, 0.704480921210089]
1222404042, 0.08159968732695994, 0.09983724809801515, 0.0769337969450224, 0.08859075518193761, 0.09131000584340572, 0.08354022900576309]
4144, 0.22336675755242635, 0.20653777973340073, 0.2030150643854944, 0.21402846769775657, 0.19187030334331637, 0.2135013669019048]
```

```
# Adding notes here as I play around with the model above
# Note 1: Regularized term = 5 -> training data loss increases significantly, same with accuracy
# Note 2: Regularized value: 0.5 ~ 1 seems to be consistent
# Note 3: Epochs = 50 significantly improves training accuracy, I find this strange, shouldn't training increase regardless of epoch number
#         and testing accuracy should decrease.
# Result: After checking testing accuracy, ~ 1-2% decrease for all models
#         Not that that accounts for much given how small the data set is & high the accuracy is. There's probably only 1 or 2 datapoints dif

print(calc_accuracy(predict(lgr_weights, x_train), y_train))
print(calc_accuracy(predict(r_lgr_weights, x_train), y_train))

print(calc_accuracy(predict(slg_weights, x_train), y_train))
print(calc_accuracy(predict(slgr_weights, x_train), y_train))

print(calc_accuracy(predict(mb_weights, x_train), y_train))
print(calc_accuracy(predict(rmb_weights, x_train), y_train))
```

```
0.91428571428571425716569365249597467482089996337890625
0.9494505494505494080925700473017059266567230224609375
0.97362637362637360904926708826678805053234100341796875
0.9472527472527472180985341765335761010646820068359375
0.97582417582417579904330295903491787612438201904296875
0.97362637362637360904926708826678805053234100341796875
```

```
# evaluate testing error of logistic regression and regularized version
```

```
print(calc_accuracy(predict(lgr_weights, x_test), y_test))
print(calc_accuracy(predict(r_lgr_weights, x_test), y_test))

print(calc_accuracy(predict(slg_weights, x_test), y_test))
print(calc_accuracy(predict(slgr_weights, x_test), y_test))

print(calc_accuracy(predict(mb_weights, x_test), y_test))
print(calc_accuracy(predict(rmb_weights, x_test), y_test))
```

```
0.9298245614035087758253439460531808435916900634765625
0.97368421052631581869007959539885632693767547607421875
0.982456140350877138445184755255468189716339111328125
0.982456140350877138445184755255468189716339111328125
0.982456140350877138445184755255468189716339111328125
0.9912280701754385692225923776277340948581695556640625
```