Assignment 7

1. **Wikipedia Wisconsin PageRank**. A dataset was created by looking at the links between Wikipedia pages with the word 'Wisconsin' in the title. The corresponding graph contains 5482 nodes (Wikipedia pages) and 41,306 edges (links between the pages).

   Two files contain the data. The edges are store in a file with two columns of integers, where the first column indicates the *from node* and the second column indicates the *to node*. A second file contains the titles of the Wikipedia pages, and their integer value.

   Use the starter script to load the edges file and create an adjacency matrix $\boldsymbol{A}$, where $\boldsymbol{A}_{i,j} = 1$ if there is a edge from node j to node i, and zero elsewhere.

   a) Write code to:

   i. ensure there are no traps by adding 0.001 to each entry of $\boldsymbol{A}$
   ii. normalize $\boldsymbol{A}$, and
   iii. use an eigen decomposition to rank the importance of the Wikipedia pages.

   b) What is the title of the page ranked 1st (i.e, the most important page)?

   c) What is the title of the page ranked 3rd?

b) Ranked first is "Wisconsin"

c) Ranked third is "Madison, Wisconsin"

2. **Gradient Descent and Logistic Regression.**

   For a binary linear classifier, the predicted class is given as $\widehat{y}_i = \text{sign}(\boldsymbol{x}_i^T \boldsymbol{w})$. Training in machine learning involves finding a $\boldsymbol{w}$ that does a good job on labeled data, and often involves solving an optimization of the form:

$$\min_{\boldsymbol{w}} \sum_i \ell_i(\boldsymbol{w}) + \lambda r(\boldsymbol{w})$$

   where $\ell_i(\boldsymbol{w})$ is the loss on the ith training example, and $r(\boldsymbol{w})$ is a regularizer. In ridge regression, the loss function is the squared error and the regularizer is the 2-norm of $\boldsymbol{w}$, and training amounts to solving the following minimization:

$$\min_{\boldsymbol{w}} \sum_{i=1}^{n} (\boldsymbol{x}_i^T \boldsymbol{w} - \boldsymbol{y}_i)^2 + \lambda ||\boldsymbol{w}||_2^2.$$

   For some classification tasks, the squared error can be a poor loss function, since easy-to-classify data points can have a large associated loss. This happens when a data point is correctly classified, but far from the decision boundary (i.e, the absolute value of $\boldsymbol{x}_i^T \boldsymbol{w}$ is large).

   In practice, an often more appropriate loss function is the logistic loss, given by

$$\ell_i(\boldsymbol{w}) = \log\left(1 + e^{-y_i \boldsymbol{x}_i^T \boldsymbol{w}}\right)$$

```
import numpy as np
from scipy.sparse import csc_matrix
from scipy.sparse.linalg import eigs

edges_file = open('wisconsin_edges.csv', "r")
nodes_file = open('wisconsin_nodes.csv', "r")

# create a dictionary where nodes_dict[i] = name of wikipedia page
nodes_dict = {}
for line in nodes_file:
    nodes_dict[int(line.split(',',1)[0].strip())] = line.split(',',1)[1].strip()

node_count = len(nodes_dict)

# create adjacency matrix
A = np.zeros((node_count, node_count))
for line in edges_file:
    from_node = int(line.split(',')[0].strip())
    to_node = int(line.split(',')[1].strip())
    A[to_node, from_node] = 1.0

# Hint -- instead of computing the entire eigen-decomposition of a matrix X using
# s, E = np.linalg.eig(A)
# you can compute just the first eigenvector with:
# s, E = eigs(csc_matrix(A), k = 1)


# i) avoid traps, add 0.001 to each entry
A += 0.001

# ii) normalize by dividing each column by the sum of each column
A = A/A.sum(axis=0,keepdims=1)

# ii) eigen decomposition
s, E = eigs(csc_matrix(A), k = 1)
E = E.real # make all the numbers real
E = abs(E)
E_sort = E[E[:, 0].argsort()][::-1]

rank_1 = np.where(E.transpose() == E_sort[0])[1]
rank_3 = np.where(E.transpose() == E_sort[2])[1]
print(rank_1)
print(rank_3)

print(nodes_dict[rank_1[0]])
print(nodes_dict[rank_3[0]])
```

```
[5089]
[1345]
"Wisconsin"
"Madison, Wisconsin"
```

a) For a binary linear classifier, explain (mathematically) why the logistic loss function does not suffer from the same problem as the squared error loss on easy to classify points.

Decision boundary value is $x_i^T w$ and actual class is y_i.
For all correctly classified points, $-y_i x_i^T w$ will be negative, thus $e^{-y_i x_i^T w}$ will be very small and the entire expression $\log(1 - e^{-y_i x_i^T w})$ will be close to $\emptyset$. Then the decision boundary will not shift by much.
Easy to classify points will have a very large negative exponent so the decision boundary will not change a lot.

b) Compute an expression for the gradient (with respect to $\boldsymbol{w}$) of the ℓ_2 regularized logistic loss:

$$\min_{\boldsymbol{w}} \sum_{i=1}^{n} \log\left(1 + e^{-y_i \boldsymbol{x}_i^T \boldsymbol{w}}\right) + \lambda ||\boldsymbol{w}||_2^2$$

$$\ell(w) = \sum_{i=1}^{n} \log(1 + e^{-y_i x_i^T w}) + \lambda \|w\|_2^2$$

$$\nabla \ell(w) = \sum_{i=1}^{n} \frac{-y_i x_i^T}{1 + e^{-y_i x_i^T w}} (e^{-y_i x_i^T w}) + \lambda 2w$$

$$= \sum_{i=1}^{n} \frac{-y_i x_i^T}{1 + e^{y_i x_i^T w}} + 2\lambda w$$

c) Use the expression for the gradient that you derived to implement gradient descent and train a classifier on the provided dataset. For simplicity, you may assume $\lambda = 1$.

d) Plot the data points (indicating their class with different colors) and plot the decision boundary. What is the error rate of your classifier on the training data?

11.45%

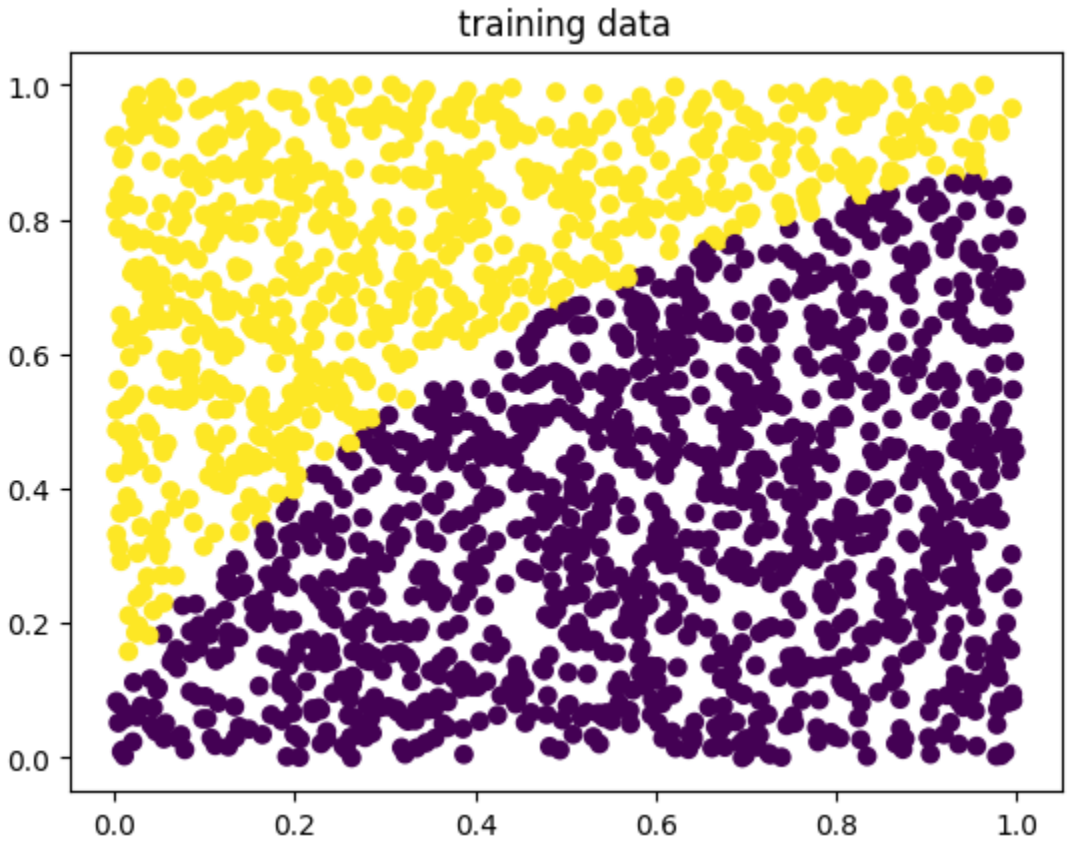
e) Train a classifier using the squared error loss, and plot the decision boundary. How does this compare with a decision boundary when trained with logistic loss?

Errors are quite similar, both are around 11.5%

f) Add 1000 easy to classify data points to the training set: more specifically, 1000 points with $y_i = -1$ and $\boldsymbol{x} = [10, 0]^T$. Re-train your classifiers and comment on the performance when trained with the logistic loss and the squared error.

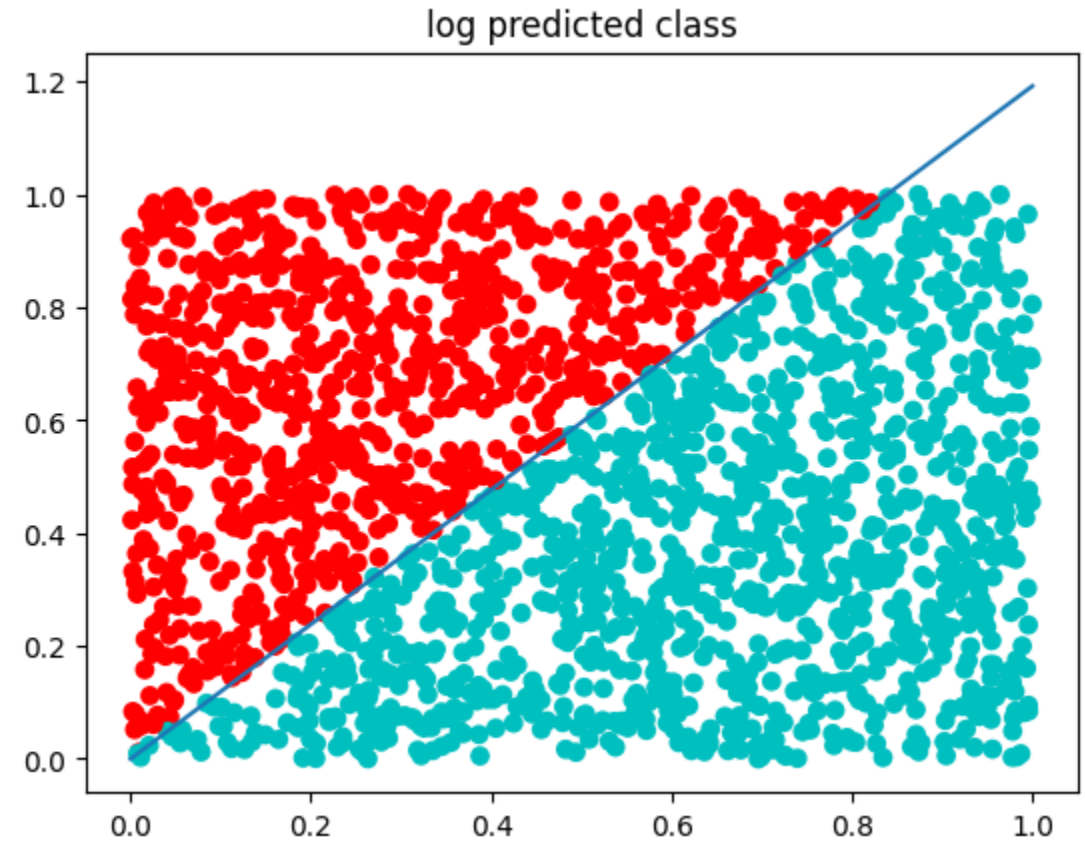
While the logistic loss error did not change by much at all, still around 11.5%, the squared error train classifier did much worse and now has about 45% error.

```
import numpy as np
import matplotlib.pyplot as plt
import pickle
import math

pkl_file = open('classifier_data.pkl', 'rb')
x_train, y_train = pickle.load(pkl_file)
n_train = np.size(y_train)

plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.title('training data')
plt.show()
```



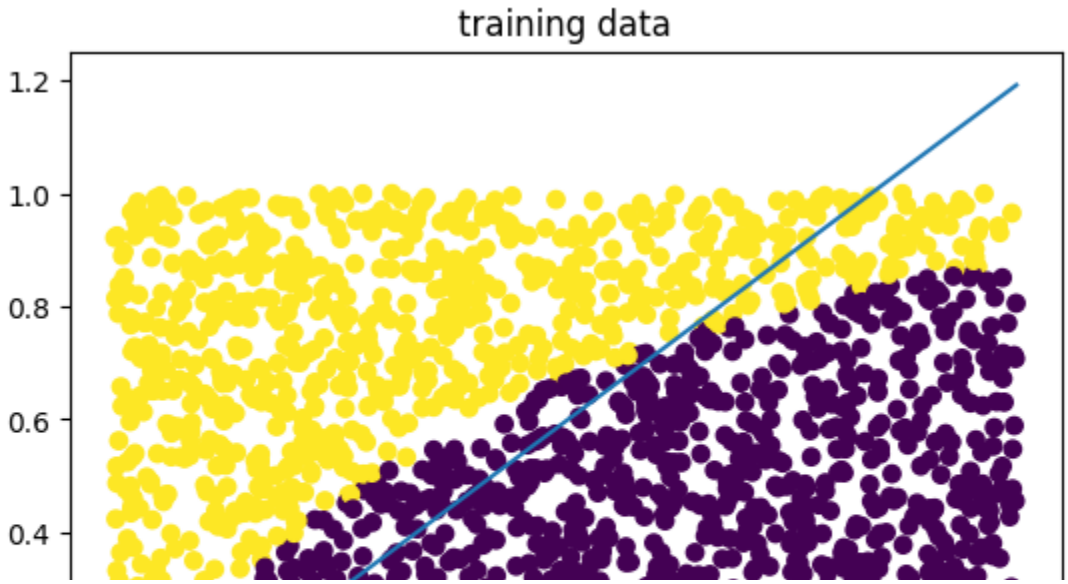

```
def log_graddescent(X,y,w_init,tau,lam,it):
  W = np.zeros((w_init.shape[0],it))
  W[:,[0]] = w_init
  gradient=0
  for k in range(it-1):
    for i in range(1,len(X)):
      gradient += -(y[i]*X[i,:].transpose()) / (1+
                                  np.exp(y[i]*X[i,:].transpose()@W[:,[k]]))
    W[:,[k+1]] = W[:,[k]]- tau*(
        np.array([gradient]).transpose() + 2*lam*W[:,[k]])
    gradient=0
  return W
```

```
w_init = np.array([[0], [0]])
W = log_graddescent(x_train, y_train, w_init, 0.01, 1, 20)
w_opt = W[:, 19]
y_hat = np.sign(x_train@w_opt)
y_hat = np.array([y_hat]).transpose()

plt.scatter(x_train[:,0],x_train[:,1],
            color=['c' if i==-1 else 'r' for i in y_hat[:,0]])
plt.plot([0, -(w_opt[0]/w_opt[1])])
plt.title('log predicted class')
plt.show()

error_vec = [0 if i[0]==i[1] else 1 for i in np.hstack((y_hat, y_train))]
print("Error Rate: ", sum(error_vec)/len(y_train))
```

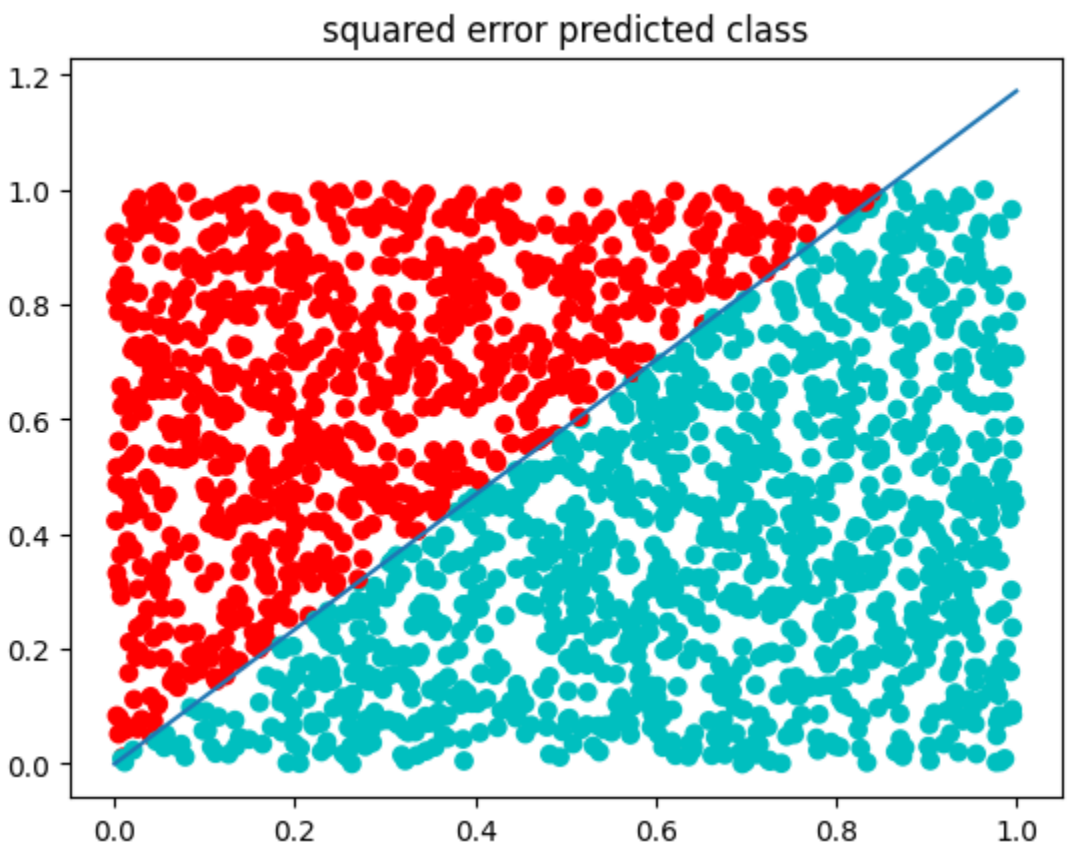



```
Error Rate:  0.1145
```

```
plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.plot([0, -(w_opt[0]/w_opt[1])])
plt.title('training data')
plt.show()
```



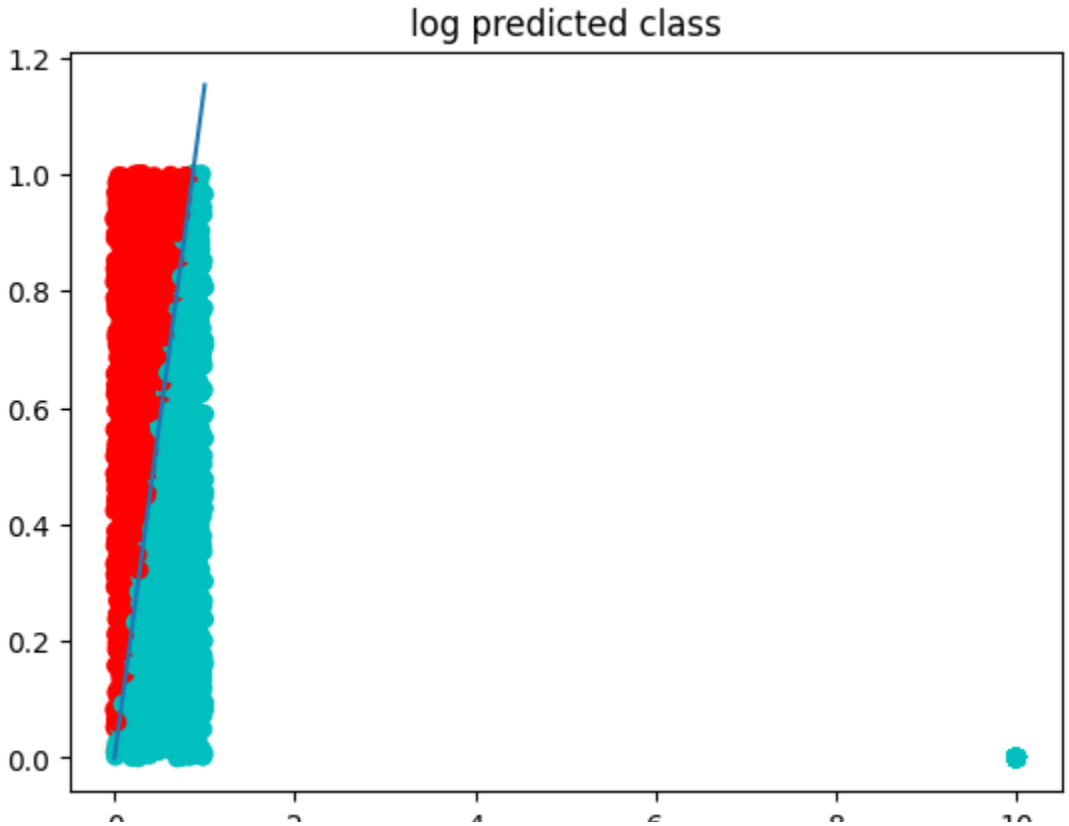

```
w_opt = np.linalg.inv(x_train.transpose()@x_train)@x_train.transpose()@y_train
y_hat = np.sign(x_train@w_opt)

plt.scatter(x_train[:,0],x_train[:,1],
            color=['c' if i==-1 else 'r' for i in y_hat[:,0]])
plt.plot([0, -(w_opt[0]/w_opt[1])])
plt.title('squared error predicted class')
plt.show()

error_vec = [0 if i[0]==i[1] else 1 for i in np.hstack((y_hat, y_train))]
print("Error Rate: ", sum(error_vec)/len(y_train))
```



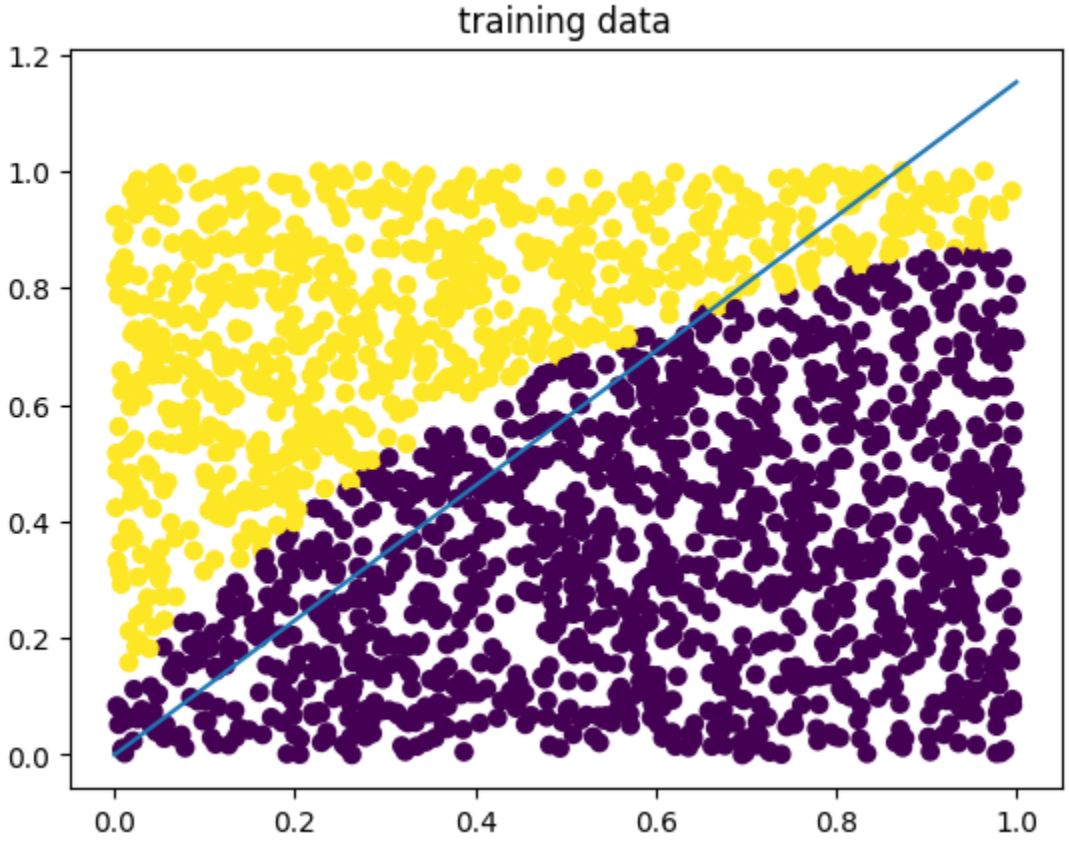

```
Error Rate:  0.114
```

```
plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.plot([0, -(w_opt[0]/w_opt[1])])
plt.title('training data')
plt.show()
```


```
new_y = np.full((1000, 1), -1)
y_train_new = np.vstack((y_train, new_y))
new_x = np.full((1000, 2), np.array([10,0]))
x_train_new = np.vstack((x_train, new_x))

w_init = np.array([[0], [0]])
W = log_graddescent(x_train_new, y_train_new, w_init, 0.01, 1, 20)

w_opt = W[:, 19]
y_hat = np.sign(x_train_new@w_opt)
y_hat = np.array([y_hat]).transpose()

plt.scatter(x_train_new[:,0],x_train_new[:,1],
            color=['c' if i==-1 else 'r' for i in y_hat[:,0]])
plt.plot([0, -(w_opt[0]/w_opt[1])])
plt.title('log predicted class')
plt.show()

error_vec = [0 if i[0]==i[1] else 1 for i in np.hstack((y_hat, y_train_new))]
print("Error Rate: ", sum(error_vec)/len(y_train))
```

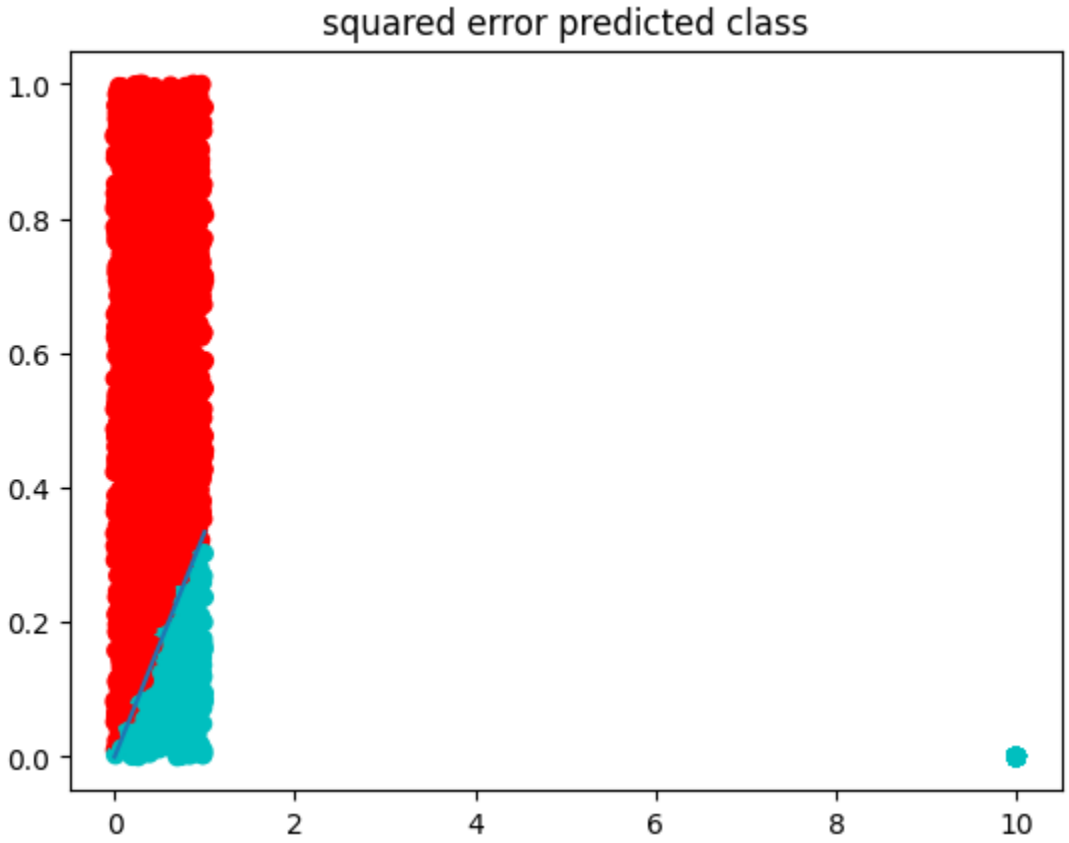



```
plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.plot([0, -(w_opt[0]/w_opt[1])])
plt.title('training data')
plt.show()
```




```
w_opt = np.linalg.inv(
```

```
    x_train_new.transpose()@x_train_new)@x_train_new.transpose()@y_train_new
y_hat = np.sign(x_train_new@w_opt)

plt.scatter(x_train_new[:,0],x_train_new[:,1],
            color=['c' if i==-1 else 'r' for i in y_hat[:,0]])
plt.plot([0, -(w_opt[0]/w_opt[1])])
plt.title('squared error predicted class')
plt.show()

error_vec = [0 if i[0]==i[1] else 1 for i in np.hstack((y_hat, y_train_new))]
print("Error Rate: ", sum(error_vec)/len(y_train))
```




```
Error Rate:  0.451
```

```
plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.plot([0, -(w_opt[0]/w_opt[1])])
plt.title('training data')
plt.show()
```