CS 189 Introduction to Machine Learning
Fall 2017 HW10

This homework is due **Monday, November 6 at 10pm.**

1 Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, "HW[n] Write-Up"
2. Submit all code needed to reproduce your results, "HW[n] Code".
3. Submit your test set evaluation results, "HW[n] Test Set".

After you've submitted your homework, be sure to watch out for the self-grade form.

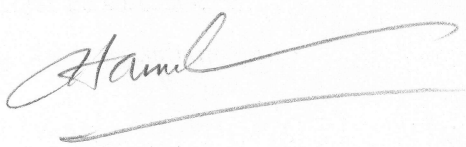
(a) Before you start your homework, write down your team. Who else did you work with on this homework? List names and email addresses. In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

None
Comments: N/A

(b) Please copy the following statement and sign next to it:

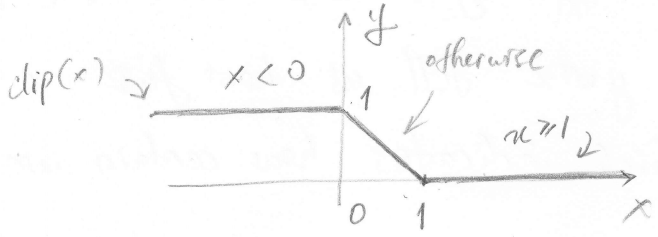
I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

I certify that all solutions are entirely in my words & that I have not looked @ another student's solutions. I have credited all external sources in this write up



Problem # 2

(a)



The function is not convex.

The convex condition is:

$$\alpha f(x_1) + (1-\alpha) f(x_2) \geq f(\alpha x_1 + (1-\alpha) x_2)$$

$\forall x_1, x_2 ; \alpha \in [0, 1]$

Take $\alpha = 0.5$, $x_1 = -1$, $x_2 = 1$

$$LHS = 0.5\, clip(-1) + 0.5\, clip(1) = 0.5 \times 1 + 0.5 \times 0 = 0.5$$

$$RHS = clip(0.5 \times (-1) + 0.5 \times 1) = clip(0) = 1 - 0 = 1$$

$LHS < RHS \rightarrow$ not convex

(b) The loss function checks the signs of y & $w^T x$. If y and $w^T x$ have opposite signs → there is a miss → penalty if they have the same signs → two scenarios:

1.) if $w^T x$ is close to 0 → penalty a little bit by $1 - y w^T x = 1 - \| w^T x \|$ (since y & $w^T x$ have same sign)

2.) If $w^T x$ is large → it is clear that it has label y → no penalty

∴ the region from $0 \rightarrow 1$ of $y w^T x$ can be considered uncertain area.

It is reasonable to look at yw^Tx because if we have only two classes, x has label y iff y & w^Tx have the same signs and the magnitude of yw^Tx tell us how far x is away from the boundary, which indicates how certain we are about the label of x.

(c)

$$R_S[w] = 0$$

$$\frac{1}{n}\sum_{i=1}^{n} \text{loss}(w^Tx_i, y_i) = 0$$

$$\sum_{i=1}^{n} \text{clip}(y_i w^T x_i) = 0 \qquad (*)$$

clip() is a positive function (see graph)

$(*) \Rightarrow$ $\text{clip}(y_i w^T x_i) = 0 \quad \forall (x_i, y_i) \in S$

$\Rightarrow$ $y_i w^T x_i \geq 1$

$\Rightarrow$ $w^Tx_i \geq 1$ or $w^Tx_i \leq -1$

i.e $|w^Tx_i| \geq 1 \quad \forall x_i$

The classification margin is the distance from the boundary to the nearest data point. Consider data point x, the distance from x to the hyperplane boundary is

$$d = \left| \frac{w^Tx}{\|w\|_2^2} \right| \quad \leftarrow \text{absolute value}$$

$$= \frac{|w^Tx|}{\|w\|_2^2} \quad \begin{matrix} \geq 1 \\ < 1 \end{matrix} \quad \Rightarrow \quad d > 1$$

(d) $E_{\mathcal{D}}[R[w]] = E_{\mathcal{D}}\left[\frac{1}{n}\sum_{1}^{n} \text{loss}(w^T x_i, y_i)\right]$

$= \frac{1}{n}\sum_{1}^{n} \underbrace{E_{\mathcal{D}}[\text{loss}(w^T x_i, y_i)]}_{R[w]}$ $\quad (x_i, y_i)$ sampled i.i.d from $\mathcal{D}$

$= \frac{1}{n}\, n\, R[w] = R[w]$

(e) $\text{Var}[R_S[w]] = \text{Var}\left[\frac{1}{n}\sum_{i=1}^{n} \text{loss}(w^T x_i, y_i)\right]$

$= \frac{1}{n^2}\sum_{i=1}^{n} \text{Var}\left(\text{clip}(y_i w^T x_i)\right)$

$\text{clip}(y_i w^T x_i) \le 1 \quad \forall \ (x_i, y_i)$

$\Rightarrow \text{Var}(\text{clip}(y_i w^T, x_i) \le 1$

$\Rightarrow \sum_{i=1}^{n} \text{Var}(\text{clip}(y_i w^T x_i)) \le n$

$\Rightarrow \text{Var}[R_S[w]] \le \frac{1}{n^2}\, n = \frac{1}{n}$

(f) It is possible. Consider S has only one point (x_1, y_1) and this point is correctly classified $\Rightarrow \text{loss}(w^T x_1, y_1) = 0$

$\Rightarrow R_S[w] = \frac{1}{n}\sum_{i=1}^{n} \text{loss}(w^T x_i, y_i) = \frac{1}{1}\text{loss}(w^T x_1, y_1) = 0$

but if the classifier is not perfect $\rightarrow$ there are some losses over $\mathcal{D}$ $\Rightarrow$ $R[w] = E_{\mathcal{D}}[\text{loss}(w^T x, y)] > 0$

since loss is a positive function

(thus its expected value is positive)

Problem #3

(a) See code attached

(b) See code attached.

If we increase the learning rate, there are two scenarios.

1.) If the learning rate is still less than the critical value, the error converges faster

2.) If the learning rate is greater than the critical value, the error blows up.

Comparison: batch GD converges faster than full GD (see plots)

(c) See code attached

Comparison: Full LS is the best if it hits all features next winner is stochastic gradient descent for all features, then full GD for all features. GDs for one feature are uncertain but stochastic is better than full (see plots)

(d) See code attached

Comparison: Kaczmarz SGD is worse than full GD for all features but it seems to be stable. (see plots)

(e) See code attached

Comparison: Basically, the non-linear cases have the same behaviors as the linear cases for all methods but with larger error initially.

Problem #4

(a) See code attached

(b) See code attached

(c) See code attached

(d) See code attached

(e) See code attached

(f) See code attached

(g) See code attached

(h) See code attached

For k small, sum works well but for k larges, QDA is better

=> Choose QDA w/ $k = 800$

Problem #5

To continue problem 2(c), assume the hyperplane not only defined by w but having bias b. that is the boundary hyperplane is given by $w^T x = b$

(a) Find the distance from a data point x^* to this plane

(b) What is the new classification margin

Solution

we know that without bias, the plane is

$$w^T x = 0 \qquad (P)$$

having unit normal $\frac{w}{\|w\|_2^2}$, and this plane going through the origin $\Rightarrow$ the distance from x^* to (P) is $\frac{w^T x^*}{\|w\|_2^2} = d$

Consider

$$w^T x = b$$

$$w_1 x_1 + w_2 x_2 + \dots + w_n x_n - w_n \cdot \frac{b}{w_n} = 0$$

$$w^T x' = 0 \qquad \text{where } x' = \begin{bmatrix} x_1 \\ \vdots \\ x_n - \frac{b}{w_n} \end{bmatrix} = x - \underbrace{\begin{bmatrix} 0 \\ 0 \\ \vdots \\ \frac{b}{w_n} \end{bmatrix}}_{\bar{x}}$$

For x^* : $x' = x^* - \bar{x}$

$\Rightarrow$ new distance : $\frac{w^T(x^* - \bar{x})}{\|w\|_2^2} = d - \underbrace{\frac{w^T \bar{x}}{\|w\|_2^2}} = d - \frac{b}{\|w\|_2^2}$

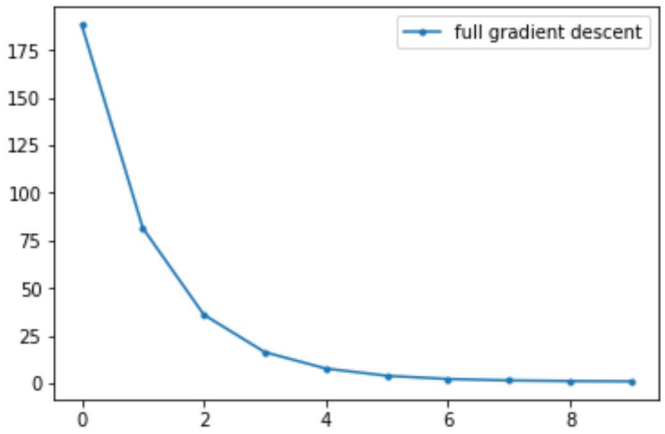
(b) the new classification margin is

$$> 1 - \frac{b}{\|w\|_2^2}$$

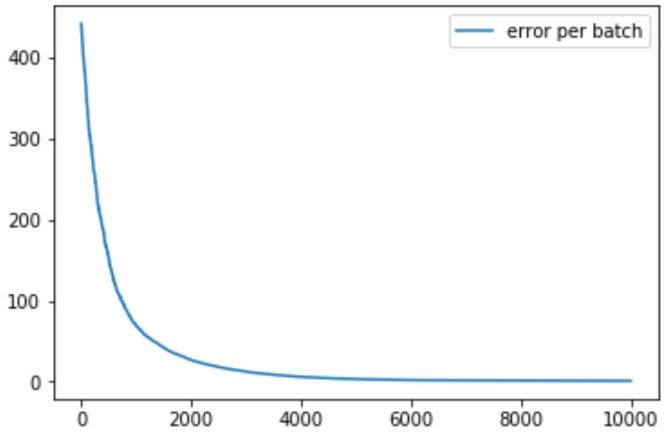
(since we proved in 2(c) that the old classification margin is > 1)

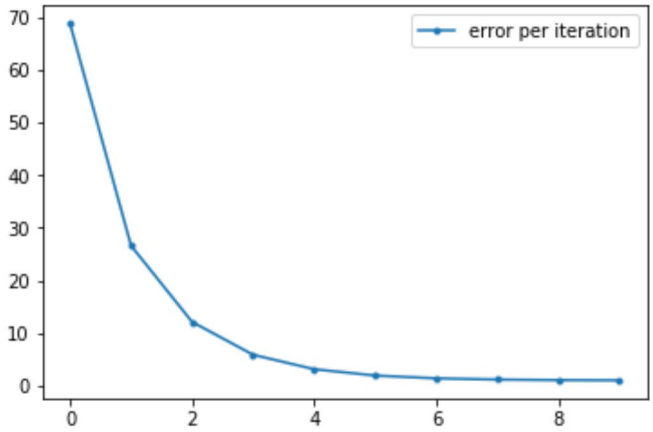
In [67]:
```
import numpy as np
import numpy.linalg as npla
import matplotlib.pyplot as plt
import scipy.io as scio
import random

DATAFILE = "gradient_descent_data.mat"

data = scio.loadmat(DATAFILE)
A = data['x']     # (1000, 2)
y = data['y']     # (1000, 1)
n, d = A.shape    # n = 1000 number of samples, d = 2 number of features
```

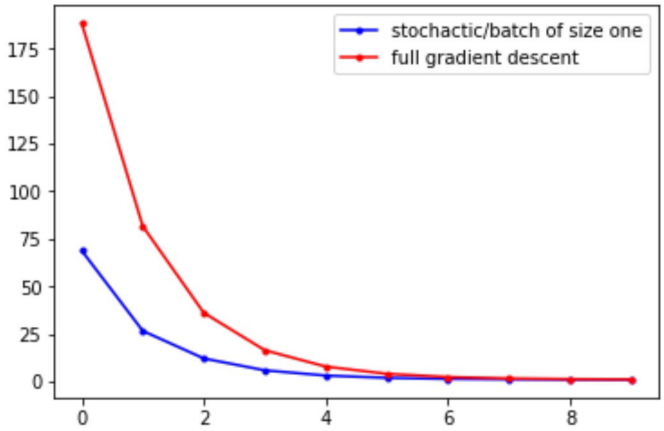
In [103]:
```
# Problem 3.a

alpha = 0.05
w = np.zeros( (d, 1), "float" )
errors_3a = np.zeros( (10, 1), "float" )
for i in range(10):
    grad_loss = 2 * A.T.dot(A).dot(w) - 2 * A.T.dot(y)
    w = w - alpha * grad_loss / (2 * n)
    errors_3a[i] = np.mean((A.dot(w) - y) ** 2)

plt.figure()
plt.plot(errors_3a, ".-", label="full gradient descent")
plt.legend(loc="best")
plt.show()
```



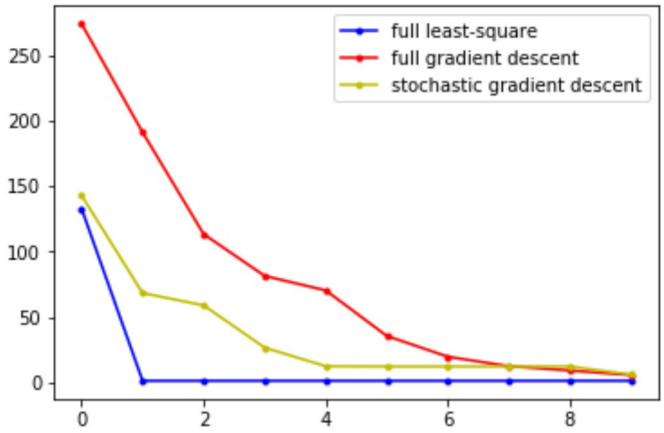
In [104]:
```
# Problem 3.b

alpha = 0.05
w = np.zeros( (d, 1), "float" )
errors_per_iter = np.zeros( (10, 1), "float" )
errors_per_batch = np.zeros( (10, n), "float" )
for i in range(10):
    for j in range(n):
        grad_loss = 2 * A[[j],:].T.dot(A[[j],:]).dot(w) - 2 * A[[j],:].T.dot(y[[
j]])
        w = w - alpha * grad_loss / (2 * n)
        errors_per_batch[i,j] = np.mean((A.dot(w) - y) ** 2)
    errors_per_iter[i] = np.mean((A.dot(w) - y) ** 2)

plt.figure()
plt.plot(errors_per_batch.flatten(), label="error per batch")
plt.legend(loc="best")
plt.figure()
plt.plot(errors_per_iter, ".-", label="error per iteration")
plt.legend(loc="best")
plt.figure()
plt.plot(errors_per_iter, "b.-", label="stochactic/batch of size one")
plt.plot(errors_3a, "r.-", label="full gradient descent")
plt.legend(loc="best")
plt.show()
```

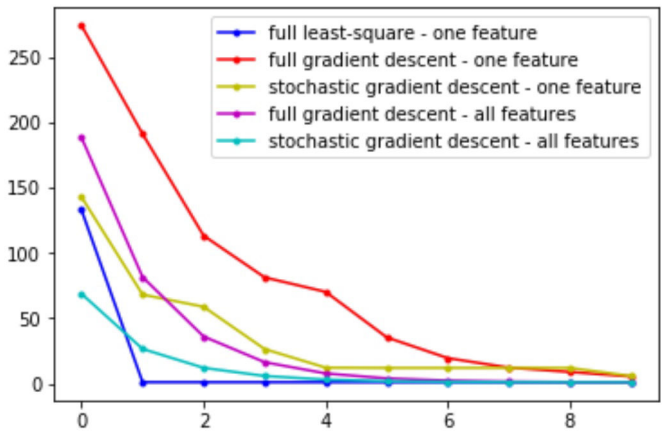

error per batch
400
300
200
100
0
0
2000
4000
6000
8000
10000


error per iteration
70
60
50
40
30
20
10
0
0
2
4
6
8

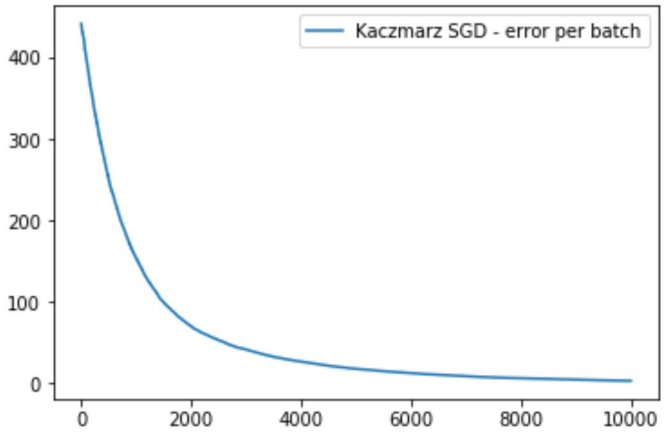

stochactic/batch of size one
full gradient descent
175
150
125
100
75
50
25
0
0
2
4
6
8

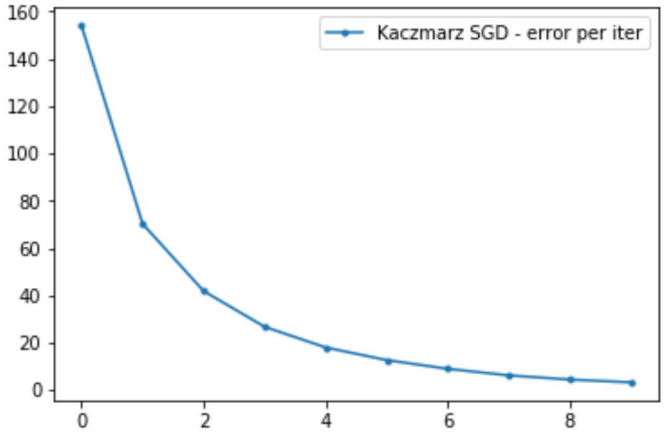
In [81]:
```
# Problem 3.c

alpha = 0.05
w1 = np.zeros( (d, 1), "float" )
w2 = np.zeros( (d, 1), "float" )
w3 = np.zeros( (d, 1), "float" )
errors1 = np.zeros( (10, 1), "float" )
errors2 = np.zeros( (10, 1), "float" )
errors3 = np.zeros( (10, 1), "float" )

for i in range(10):
    r = random.random()
    k = 0 if r < 0.5 else 1
    # full least-square
    w1[k] = npla.inv( A[:,[k]].T.dot(A[:,[k]]) ).dot( A[:,[k]].T.dot(y) )
    errors1[i] = np.mean((A.dot(w1) - y) ** 2)
    # full gradient descent
    grad_loss = 2 * A.T.dot(A).dot(w2) - 2 * A.T.dot(y)
    w2[k] = w2[k] - alpha * grad_loss[k] / (2 * n)
    errors2[i] = np.mean((A.dot(w2) - y) ** 2)
    # stochastic gradient descent
    for j in range(n):
        grad_loss = 2 * A[[j],:].T.dot(A[[j],:]).dot(w3) - 2 * A[[j],:].T.dot(y[
[j]])
        w3[k] = w3[k] - alpha * grad_loss[k] / (2 * n)
    errors3[i] = np.mean((A.dot(w3) - y) ** 2)

plt.figure()
plt.plot(errors1, "b.-", label="full least-square")
plt.plot(errors2, "r.-", label="full gradient descent")
plt.plot(errors3, "y.-", label="stochastic gradient descent")
plt.legend(loc="best")
plt.figure()
plt.plot(errors1, "b.-", label="full least-square - one feature")
plt.plot(errors2, "r.-", label="full gradient descent - one feature")
plt.plot(errors3, "y.-", label="stochastic gradient descent - one feature")
plt.plot(errors_3a, "m.-", label="full gradient descent - all features")
plt.plot(errors_per_iter, "c.-", label="stochastic gradient descent - all featur
es")
plt.legend(loc="best")
plt.show()
```

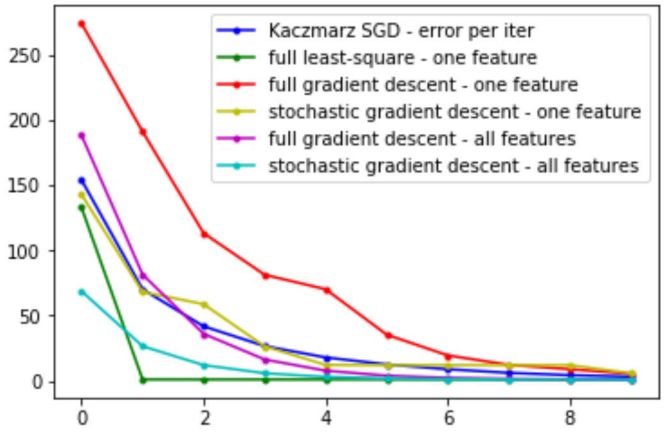

full least-square
full gradient descent
stochastic gradient descent
250
200
150
100
50
0
0
2
4
6
8


full least-square - one feature
full gradient descent - one feature
stochastic gradient descent - one feature
full gradient descent - all features
stochastic gradient descent - all features
250
200
150
100
50
0
0
2
4
6
8

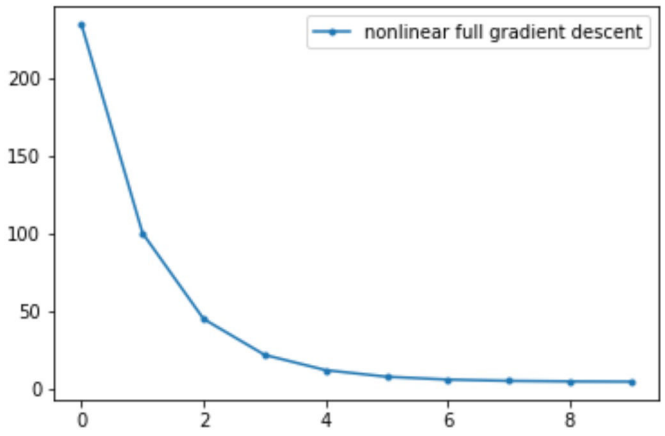
In [82]:
```
# Problem 3.d

w = np.zeros( (d, 1), "float" )
errors_per_iter_ aczmarz = np.zeros( (10, 1), "float" )
errors_per_batch_ aczmarz = np.zeros( (10, n), "float" )

alpha_list = []
prob_list = []
cummulative_prob_list = []
A_Froberius = np.sum(A ** 2)
for j in range(n):
    Aj_Froberius = np.sum( A[j, :] ** 2 )
    alpha_list.append(1 / Aj_Froberius)
    prob_list.append(Aj_Froberius / A_Froberius)
    if j == 0:
        cummulative_prob_list.append(Aj_Froberius / A_Froberius)
    else:
        cummulative_prob_list.append(cummulative_prob_list[j - 1]   Aj_Froberius
/ A_Froberius)

for i in range(10):
    for j in range(n):
        r = random.random()
        index = 0
          ile cummulative_prob_list[index] < r:
            index  = 1
        grad_loss = 2 * A[[index],:].T.dot(A[[index],:]).dot(w) - 2 * A[[index],
:].T.dot(y[[index]])
        w = w - alpha_list[index] * grad_loss / (2 * n)
        errors_per_batch_ aczmarz[i,j] = np.mean((A.dot(w) - y) ** 2)
    errors_per_iter_ aczmarz[i] = np.mean((A.dot(w) - y) ** 2)

plt.figure()
plt.plot(errors_per_batch_ aczmarz.flatten(), label=" aczmarz   D - error per ba
tch")
plt.legend(loc="best")
plt.figure()
plt.plot(errors_per_iter_ aczmarz, ".-", label=" aczmarz   D - error per iter")
plt.legend(loc="best")
plt.figure()
plt.plot(errors_per_iter_ aczmarz, "b.-", label=" aczmarz   D - error per iter")
plt.plot(errors1, "g.-", label="full least-square - one feature")
plt.plot(errors2, "r.-", label="full gradient descent - one feature")
plt.plot(errors3, "y.-", label="stochastic gradient descent - one feature")
plt.plot(errors_3a, "m.-", label="full gradient descent - all features")
plt.plot(errors_per_iter, "c.-", label="stochastic gradient descent - all featur
es")
plt.legend(loc="best")
plt.show()
```


Kaczmarz SGD - error per batch
400
300
200
100
0
0
2000
4000
6000
8000
10000

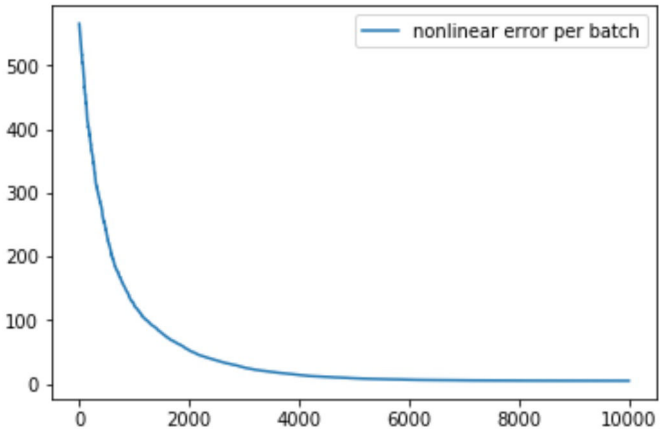

Kaczmarz SGD - error per iter
160
140
120
100
80
60
40
20
0
0
2
4
6
8

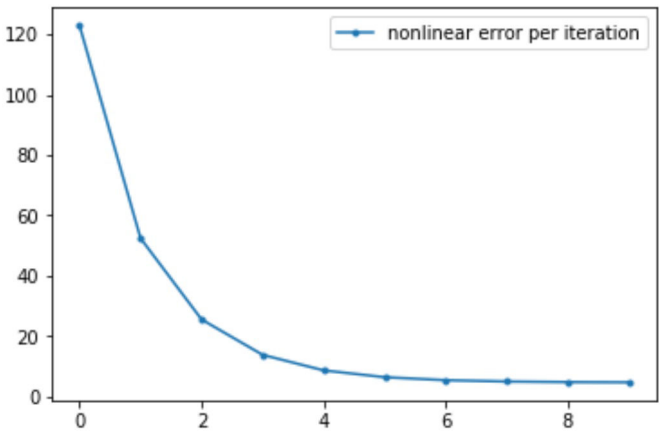

Kaczmarz SGD - error per iter
full least-square - one feature
full gradient descent - one feature
stochastic gradient descent - one feature
full gradient descent - all features
stochastic gradient descent - all features
250
200
150
100
50
0
0
2
4
6
8

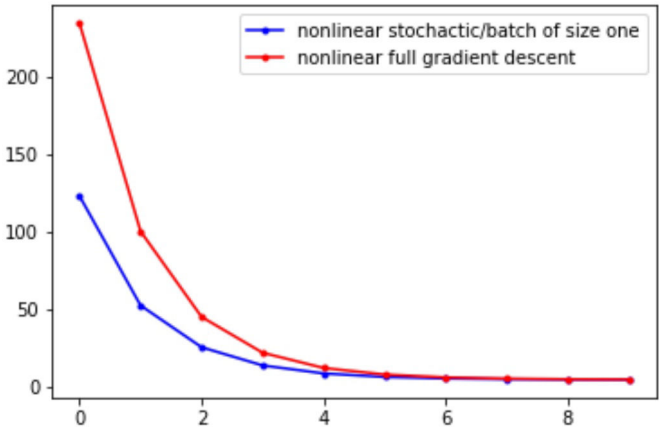
In [96]:
```
# Problem 3.e

y_nonlinear = y + 0.1 * A[:, [1]] ** 3

# Repeat Problem 3.a

alpha = 0.05
w = np.zeros( (d, 1), "float" )
errors_3a_nonlinear = np.zeros( (10, 1), "float" )
for i in range(10):
    grad_loss = 2 * A.T.dot(A).dot(w) - 2 * A.T.dot(y_nonlinear)
    w = w - alpha * grad_loss / (2 * n)
    errors_3a_nonlinear[i] = np.mean((A.dot(w) - y_nonlinear) ** 2)

plt.figure()
plt.plot(errors_3a_nonlinear, ".-", label="nonlinear full gradient descent")
plt.legend(loc="best")
plt.show()
```



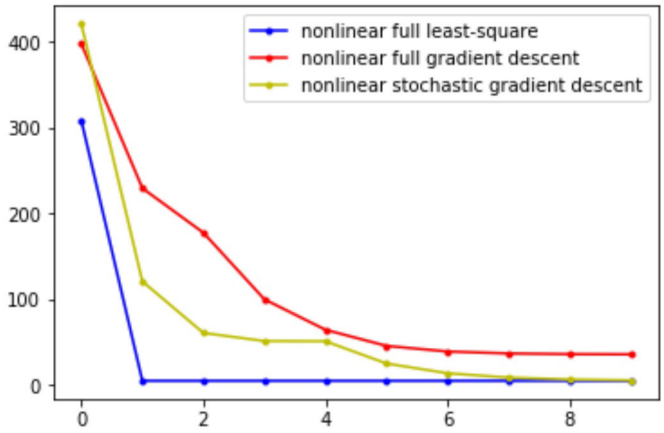
In [98]:
```
# Repeat Problem 3.b

alpha = 0.05
w = np.zeros( (d, 1), "float" )
errors_per_iter_nonlinear = np.zeros( (10, 1), "float" )
errors_per_batch_nonlinear = np.zeros( (10, n), "float" )
for i in range(10):
    for j in range(n):
        grad_loss = 2 * A[[j],:].T.dot(A[[j],:]).dot(w) - 2 * A[[j],:].T.dot(y_n
onlinear[[j]])
        w = w - alpha * grad_loss / (2 * n)
        errors_per_batch_nonlinear[i,j] = np.mean((A.dot(w) - y_nonlinear) ** 2)
    errors_per_iter_nonlinear[i] = np.mean((A.dot(w) - y_nonlinear) ** 2)

plt.figure()
plt.plot(errors_per_batch_nonlinear.flatten(), label="nonlinear error per batch"
)
plt.legend(loc="best")
plt.figure()
plt.plot(errors_per_iter_nonlinear, ".-", label="nonlinear error per iteration")
plt.legend(loc="best")
plt.figure()
plt.plot(errors_per_iter_nonlinear, "b.-", label="nonlinear stochactic/batch of
size one")
plt.plot(errors_3a_nonlinear, "r.-", label="nonlinear full gradient descent")
plt.legend(loc="best")
plt.show()
```

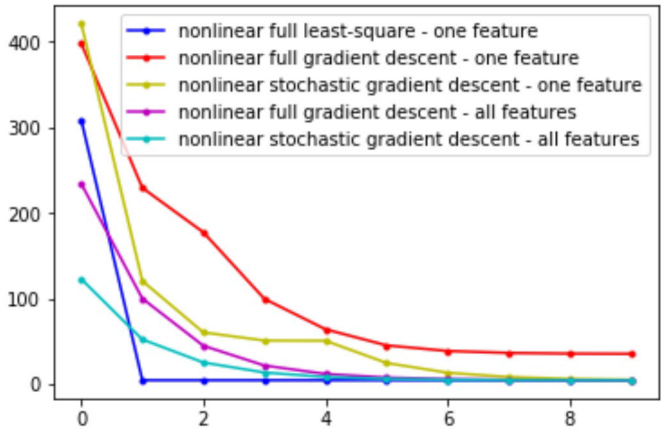

nonlinear error per batch
500
400
300
200
100
0
0
2000
4000
6000
8000
10000


nonlinear error per iteration
120
100
80
60
40
20
0
0
2
4
6
8

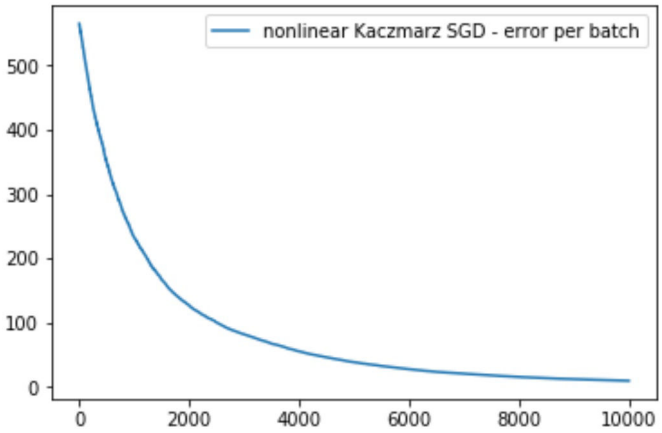

nonlinear stochactic/batch of size one
nonlinear full gradient descent
200
150
100
50
0
0
2
4
6
8

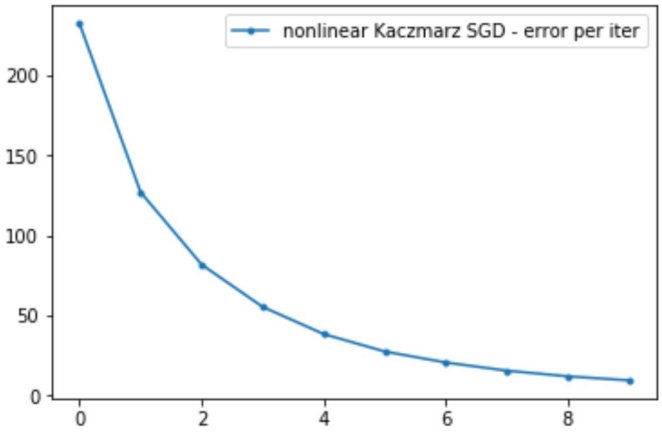
In [99]:
```
# Repeat Problem 3.c

alpha = 0.05
w1 = np.zeros( (d, 1), "float" )
w2 = np.zeros( (d, 1), "float" )
w3 = np.zeros( (d, 1), "float" )
errors1_nonlinear = np.zeros( (10, 1), "float" )
errors2_nonlinear = np.zeros( (10, 1), "float" )
errors3_nonlinear = np.zeros( (10, 1), "float" )

for i in range(10):
    r = random.random()
    k = 0 if r < 0.5 else 1
    # full least-square
    w1[k] = npla.inv( A[:,[k]].T.dot(A[:,[k]]) ).dot( A[:,[k]].T.dot(y_nonlinear
) )
    errors1_nonlinear[i] = np.mean((A.dot(w1) - y_nonlinear) ** 2)
    # full gradient descent
    grad_loss = 2 * A.T.dot(A).dot(w2) - 2 * A.T.dot(y_nonlinear)
    w2[k] = w2[k] - alpha * grad_loss[k] / (2 * n)
    errors2_nonlinear[i] = np.mean((A.dot(w2) - y_nonlinear) ** 2)
    # stochastic gradient descent
    for j in range(n):
        grad_loss = 2 * A[[j],:].T.dot(A[[j],:]).dot(w3) - 2 * A[[j],:].T.dot(y_
nonlinear[[j]])
        w3[k] = w3[k] - alpha * grad_loss[k] / (2 * n)
    errors3_nonlinear[i] = np.mean((A.dot(w3) - y_nonlinear) ** 2)

plt.figure()
plt.plot(errors1_nonlinear, "b.-", label="nonlinear full least-square")
plt.plot(errors2_nonlinear, "r.-", label="nonlinear full gradient descent")
plt.plot(errors3_nonlinear, "y.-", label="nonlinear stochastic gradient descent"
)
plt.legend(loc="best")
plt.figure()
plt.plot(errors1_nonlinear, "b.-", label="nonlinear full least-square - one feat
ure")
plt.plot(errors2_nonlinear, "r.-", label="nonlinear full gradient descent - one
feature")
plt.plot(errors3_nonlinear, "y.-", label="nonlinear stochastic gradient descent
- one feature")
plt.plot(errors_3a_nonlinear, "m.-", label="nonlinear full gradient descent - al
l features")
plt.plot(errors_per_iter_nonlinear, "c.-", label="nonlinear stochastic gradient
descent - all features")
plt.legend(loc="best")
plt.show()
```

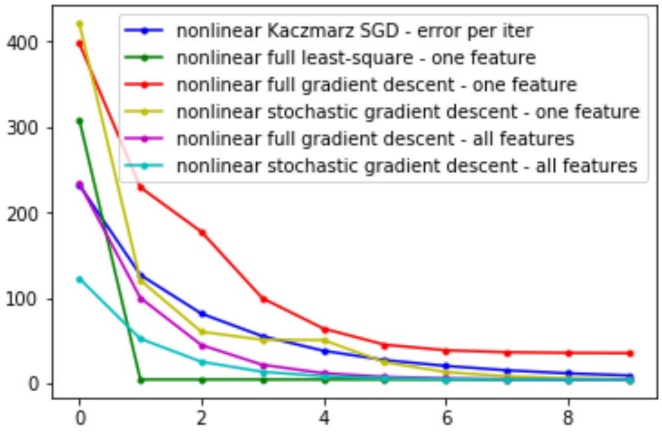

nonlinear full least-square
nonlinear full gradient descent
nonlinear stochastic gradient descent
400
300
200
100
0
0
2
4
6
8


nonlinear full least-square - one feature
nonlinear full gradient descent - one feature
nonlinear stochastic gradient descent - one feature
nonlinear full gradient descent - all features
nonlinear stochastic gradient descent - all features
400
300
200
100
0
0
2
4
6
8

In [100]:
```
# Repeat Problem 3.d

w = np.zeros( (d, 1), "float" )
errors_per_iter_Kaczmarz_nonlinear = np.zeros( (10, 1), "float" )
errors_per_batch_Kaczmarz_nonlinear = np.zeros( (10, n), "float" )

alpha_list = []
prob_list = []
cummulative_prob_list = []
A_Froberius = np.sum(A ** 2)
for j in range(n):
    Aj_Froberius = np.sum( A[j, :] ** 2 )
    alpha_list.append(1 / Aj_Froberius)
    prob_list.append(Aj_Froberius / A_Froberius)
    if j == 0:
        cummulative_prob_list.append(Aj_Froberius / A_Froberius)
    else:
        cummulative_prob_list.append(cummulative_prob_list[j - 1] + Aj_Froberius
/ A_Froberius)

for i in range(10):
    for j in range(n):
        r = random.random()
        index = 0
        while cummulative_prob_list[index] < r:
            index += 1
        grad_loss = 2 * A[[index],:].T.dot(A[[index],:]).dot(w) - 2 * A[[index],
:].T.dot(y_nonlinear[[index]])
        w = w - alpha_list[index] * grad_loss / (2 * n)
        errors_per_batch_Kaczmarz_nonlinear[i,j] = np.mean((A.dot(w) - y_nonline
ar) ** 2)
    errors_per_iter_Kaczmarz_nonlinear[i] = np.mean((A.dot(w) - y_nonlinear) **
2)

plt.figure()
plt.plot(errors_per_batch_Kaczmarz_nonlinear.flatten(), label="nonlinear Kaczmar
z SGD - error per batch")
plt.legend(loc="best")
plt.figure()
plt.plot(errors_per_iter_Kaczmarz_nonlinear, ".-", label="nonlinear Kaczmarz SGD
- error per iter")
plt.legend(loc="best")
plt.figure()
plt.plot(errors_per_iter_Kaczmarz_nonlinear, "b.-", label="nonlinear Kaczmarz SG
D - error per iter")
plt.plot(errors1_nonlinear, "g.-", label="nonlinear full least-square - one feat
ure")
plt.plot(errors2_nonlinear, "r.-", label="nonlinear full gradient descent - one
feature")
plt.plot(errors3_nonlinear, "y.-", label="nonlinear stochastic gradient descent
- one feature")
plt.plot(errors_3a_nonlinear, "m.-", label="nonlinear full gradient descent - al
l features")
plt.plot(errors_per_iter_nonlinear, "c.-", label="nonlinear stochastic gradient
descent - all features")
plt.legend(loc="best")
plt.show()
```

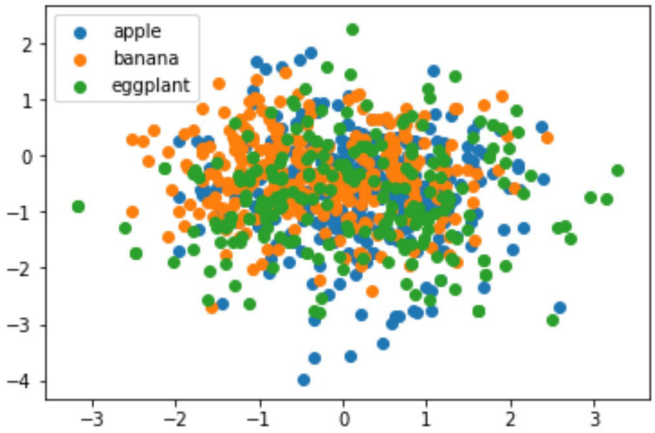

nonlinear Kaczmarz SGD - error per batch
500
400
300
200
100
0
0
2000
4000
6000
8000
10000

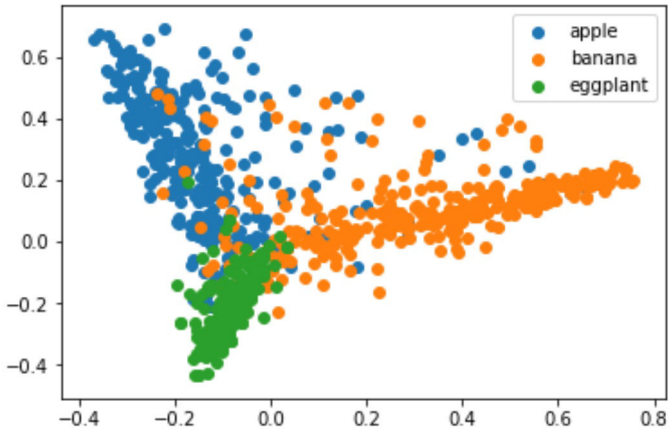

nonlinear Kaczmarz SGD - error per iter
200
150
100
50
0
0
2
4
6
8

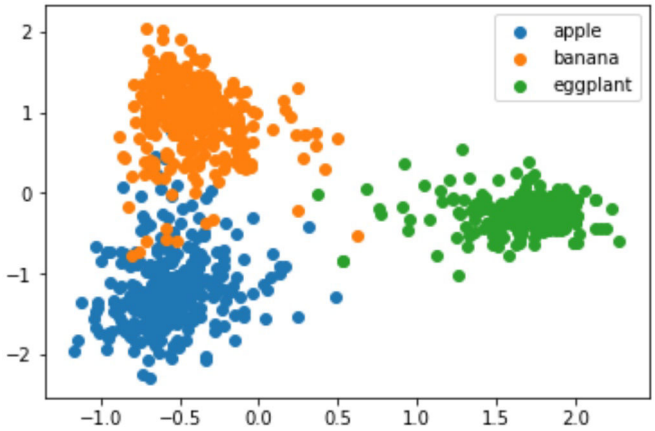

nonlinear Kaczmarz SGD - error per iter
nonlinear full least-square - one feature
nonlinear full gradient descent - one feature
nonlinear stochastic gradient descent - one feature
nonlinear full gradient descent - all features
nonlinear stochastic gradient descent - all features
400
300
200
100
0
0
2
4
6
8

In [1]:
```
# utils.py

from numpy.random import uniform
import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

from sklearn.preprocessing import StandardScaler


def create_one_hot_label(Y,N_C):
        '''
        Input
        Y: list of class labels (int)
        N_C: Number of Classes

        Returns
        List of one hot arrays with dimension N_C

        '''

        y_one_hot = []
        for y in Y:

                one_hot_label = np.zeros(N_C)

                one_hot_label[y] = 1.0
                y_one_hot.append(one_hot_label)

        return  y_one_hot


def subtract_mean_from_data(X,Y):
        '''
        Input
        X: List of data points
        Y: list of one hot class labels

        Returns
        X and Y with mean subtracted

        '''

        ss_x = StandardScaler(with_std = False)
        ss_y = StandardScaler(with_std = False)

        ss_x.fit(X)
        X = ss_x.transform(X)

        ss_y.fit(Y)
        Y = ss_y.transform(Y)

        return X,Y


def compute_covariance_matrix(X,Y):
        '''
        Input
        X: List of data points
        Y: list of one hot class labels
```

In [2]:
```
# projection.py

from numpy.random import uniform
from numpy.random import randn
import random
import time

import matplotlib.pyplot as plt

from scipy.linalg import eig
from scipy.linalg import sqrtm
from numpy.linalg import inv
from numpy.linalg import svd

from utils import create_one_hot_label
from utils import subtract_mean_from_data
from utils import compute_covariance_matrix

import numpy as np
import numpy.linalg as LA

import sys
from numpy.linalg import svd


class Project2D():

        '''
        Class to draw projection on 2D scatter space
        '''

        def __init__(self,projection, clss_labels):

                self.proj = projection
                self.clss_labels = clss_labels


        def project_data(self,X,Y,white=None):

                '''
                Takes list of state space and class labels
                State space should be 2D
                Labels shoud be int
                '''

                p_a = []
                p_b = []
                p_c = []

                # Project all Data
                proj = np.matmul(self.proj,white)

                X_P = np.matmul(proj,np.array(X).T)

                for i in range(len(Y)):

                        if Y[i] == 0:
                                p_a.append(X_P[:,i])
                        elif Y[i] == 1:
                                p_b.append(X_P[:,i])
                        else:
                                p_c.append(X_P[:,i])
```


apple
banana
eggplant
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3


apple
banana
eggplant
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8


apple
banana
eggplant
2
1
0
−1
−2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

In [3]:
```
# Ridge_Model

from numpy.random import uniform
import random
import time

import numpy as np
import numpy.linalg as LA

import sys

from sklearn.linear_model import Ridge

from utils import create_one_hot_label


class Ridge_Model():

        def __init__(self,class_labels):

                ###RIDGE HYPERPARAMETER
                self.lmda = 1.0
                self.class_labels = class_labels
                self.ridge_model = Ridge(self.lmda)


        def train_model(self,X,Y):
                ''''
                FILL IN CODE TO TRAIN MODEL
                MAKE SURE TO ADD HYPERPARAMTER TO MODEL

                '''

                X = np.array(X)
                y_one_hot = create_one_hot_label(Y, len(self.class_labels))
                self.ridge_model.fit(X, y_one_hot)


        def eval(self,x):
                ''''
                Fill in code to evaluate model and return a prediction
                Prediction should be an integer specifying a class
                '''
                x = x.reshape(1, -1)
                y = self.ridge_model.predict(x)
                return np.argmax(y)
```

In [ ]:
```
# LDA

import random
import time

import glob
import os
import pickle
import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys
from numpy.linalg import inv
from numpy.linalg import det
from sklearn.s m import  inearS C
from projection import Project2 , Projections

from utils import subtract_mean_from_data
from utils import compute_covariance_matrix


class LDA_Model():

        def __init__(self,class_labels):

                ###SCALE AN IDENTITY MATRIX BY THIS TERM AND ADD TO COMPUTED COV
ARIANCE MATRIX TO PREVENT IT BEING SINGULAR ###
                self.reg_cov = 0.001
                self.N M_C  SS S = len(class_labels)


        def train_model(self,X,Y):
                ''''
                FILL IN CODE TO TRAIN MODEL
                MAKE SURE TO ADD HYPERPARAMTER TO MODEL

                '''
                ps = [ [] for j in range(self.N M_C  SS S) ]
                for i, y in enumerate(Y):
                        ps[y].append(X[i])

                self.mean_list = []
                for lst in ps:
                        self.mean_list.append( np.mean(np.array(lst), axis=0) )

                Sigma_XX = compute_covariance_matrix(X, X)
                Sigma_XX  = self.reg_cov   np.identity(Sigma_XX.shape[0])
                self.Sigma_inv = inv(Sigma_XX)


        def eval(self,x):
                ''''
                Fill in code to evaluate model and return a prediction
                Prediction should be an integer specifying a class
                '''
                x = x.reshape(1, -1)
                y =
                for i in range(self.N M_C  SS S):
                        x_demeaned = x - self.mean_list[i]
                        f = - x_demeaned.dot(self.Sigma_inv).dot(x_demeaned.T)
```

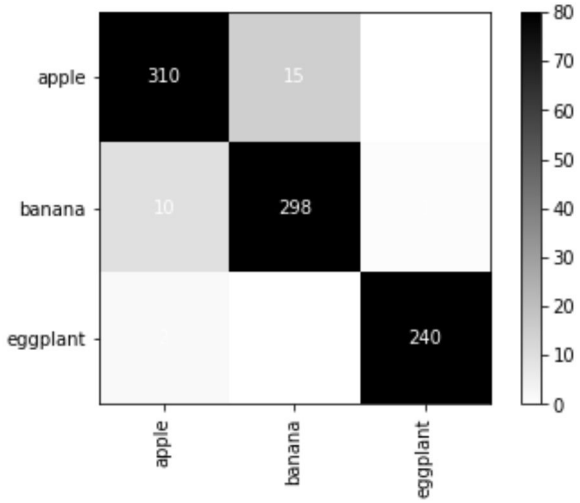
In [ ]:
```
# QDA

import random
import time

import numpy as np
import numpy.linalg as LA

from numpy.linalg import inv
from numpy.linalg import det

from projection import Project2 , Projections

from utils import subtract_mean_from_data
from utils import compute_covariance_matrix


class  DA_Model():

        def __init__(self,class_labels):

                ###SCALE AN IDENTITY MATRIX BY THIS TERM AND ADD TO COMPUTED COV
ARIANCE MATRIX TO PREVENT IT BEING SINGULAR ###
                self.reg_cov = 0.01
                self.N M_C  SS S = len(class_labels)


        def train_model(self,X,Y):
                ''''
                FILL IN CODE TO TRAIN MODEL
                MAKE SURE TO ADD HYPERPARAMTER TO MODEL

                '''
                ps = [ [] for j in range(self.N M_C  SS S) ]
                for i, y in enumerate(Y):
                        ps[y].append(X[i])

                self.mean_list = []
                self.Sigma_inv_list = []
                for lst in ps:
                        self.mean_list.append( np.mean(np.array(lst), axis=0) )
                        Sigma_XX = compute_covariance_matrix(lst, lst)
                        Sigma_XX  = self.reg_cov   np.identity(Sigma_XX.shape[0]
)
                        self.Sigma_inv_list.append(inv(Sigma_XX))


        def eval(self,x):
                ''''
                Fill in code to evaluate model and return a prediction
                Prediction should be an integer specifying a class
                '''
                x = x.reshape(1, -1)
                y =
                for i in range(self.N M_C  SS S):
                        x_demeaned = x - self.mean_list[i]
                        f = - x_demeaned.dot(self.Sigma_inv_list[i]).dot(x_demea
ned.T)
                        y[i] = f.flatten()[0]
                return max(y,  ey=lambda x: y[x])
```

In [ ]:
```
# SVM

from numpy.random import uniform
import random
import time

import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys

from sklearn.s m import  inearS C
from projection import Project2 , Projections

from utils import create_one_hot_label


class   M_Model():

        def __init__(self,class_labels,projection=None):

                ###SLACK HYPERPARAMETER
                self.C = 1.0
                self.class_labels = class_labels
                self.svc_model =  inearS C(C=self.C)


        def train_model(self,X,Y):
                ''''
                FILL IN CODE TO TRAIN MODEL
                MAKE SURE TO ADD HYPERPARAMTER TO MODEL

                '''
                X = np.array(X)
                self.svc_model.fit(X, Y)


        def eval(self,x):
                ''''
                Fill in code to evaluate model and return a prediction
                Prediction should be an integer specifying a class
                '''
                x = x.reshape(1, -1)
                y = self.svc_model.predict(x)
                return y[0]
```

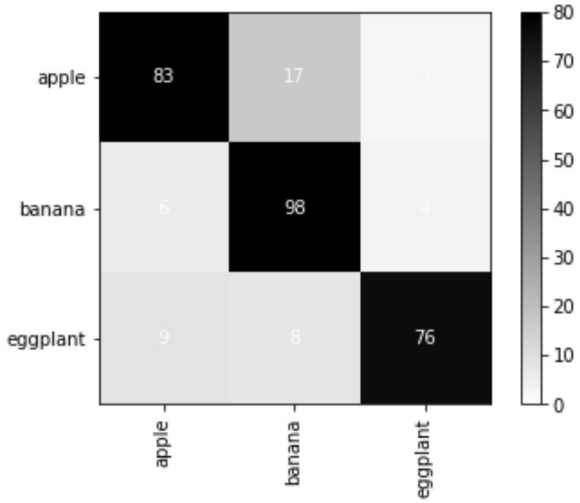
In [17]:
```
# confusion_mat.py

from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import random
import IPython

def main():
    """
    Result
    Plot RANDOM confusion matrix (true labels vs. predicted labels)
    """
    true_labels = [random.randint(1, 10) for i in range(100)]
    predicted_labels = [random.randint(1, 10) for i in range(100)]

    # Plot confusion matrix (true labels vs. predicted labels)
    plot = getConfusionMatrixPlot(true_labels, predicted_labels)
    plot.show()


def getConfusionMatrix(true_labels, predicted_labels):
    """
    Input
    true_labels: actual labels
    predicted_labels: model's predicted labels

    Output
    cm: confusion matrix (true labels vs. predicted labels)
    """

    # Generate confusion matrix using sklearn.metrics
    cm = confusion_matrix(true_labels, predicted_labels)
    return cm


def plotConfusionMatrix(cm, alphabet):
    """
    Input
    cm: confusion matrix (true labels vs. predicted labels)
    alphabet: names of class labels

    Output
    Plot confusion matrix (true labels vs. predicted labels)
    """

    fig = plt.figure()
    plt.clf()                       # Clear plot
    ax = fig.add_subplot(111)       # Add 1x1 grid, first subplot
    ax.set_aspect(1)
    res = ax.imshow(cm, cmap=plt.cm.binary,
                    interpolation='nearest', vmin=0, vmax=80)

    plt.colorbar(res)               # Add color bar

    width = len(cm)                 # Width of confusion matrix
    height = len(cm[0])             # Height of confusion matrix

    # Annotate confusion entry with numeric value
    for x in range(width):
        for y in range(height):
            ax.annotate(str(cm[x][y]), xy=(y, x), horizontalalignment='center',
                        verticalalignment='center', color=getFontColor(cm[x][y])
```

In [7]:
```
# linear_classification.py

from numpy.random import uniform
import random
import time

import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys

from projection import Project2D, Projections
from confusion_mat import getConfusionMatrixPlot

from ridge_model import Ridge_Model
from  da_model import  DA_Model
from lda_model import LDA_Model
from svm_model import SVM_Model


CLASS_LA ELS = ['apple','banana','eggplant']


class Model():
        """ Generic wrapper for specific model instance. """

        def __init__(self, model):
                """ Store specific pre-initialized model instance. """

                self.model = model


        def train_model(self,X,Y):
                """ Train using specific model's training function. """

                self.model.train_model(X,Y)


        def test_model(self,X,Y):
                """ Test using specific model's eval function. """

                labels = []                                             # List o
f actual labels
                p_labels = []                                   # List of model'
s predictions
                success = 0                                             # Number
of correct predictions
                total_count = 0                                 # Number of imag
es

                for i in range(len(X)):

                        x = X[i]                                        # Test i
nput
                        y = Y[i]                                        # Actual
label
                        y_ = self.model.eval(x)         # Model's prediction
                        labels.append(y)
                        p_labels.append(y_)
```

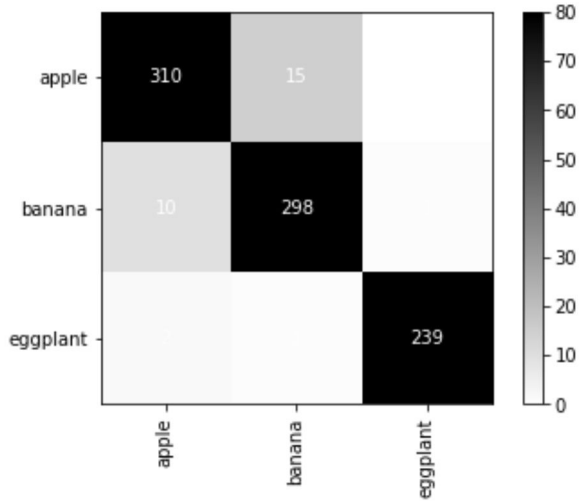
```
[[310  15   0]
 [ 10 2 8   1]
 [  2   0 240]]
```



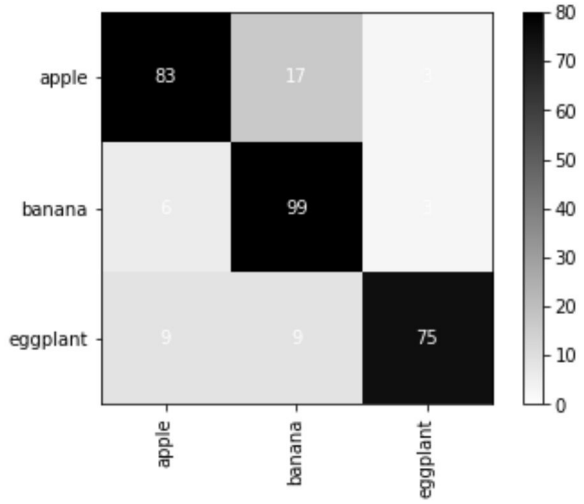

```
[[83 17  3]
 [ 6  8  4]
 [    8 76]]
```



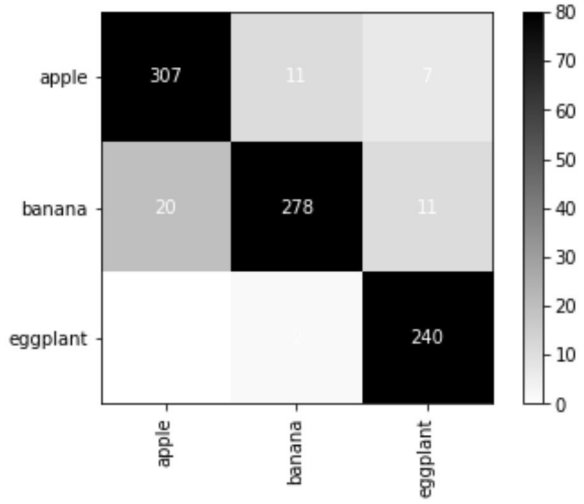

```
[[310  15   0]
 [ 10 2 8   1]
 [  2   1 23 ]]
```



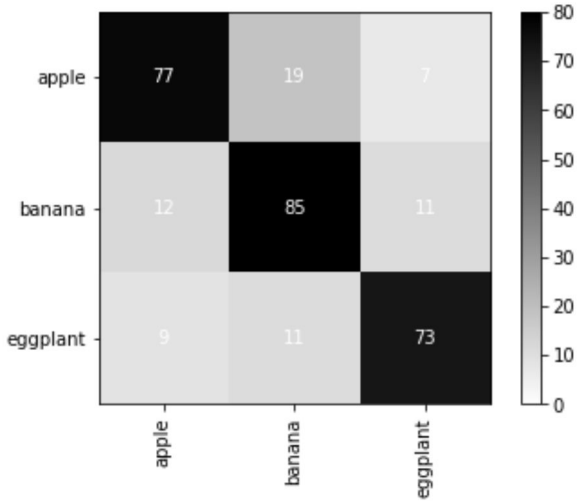

```
[[83 17  3]
 [ 6     3]
 [       75]]
```



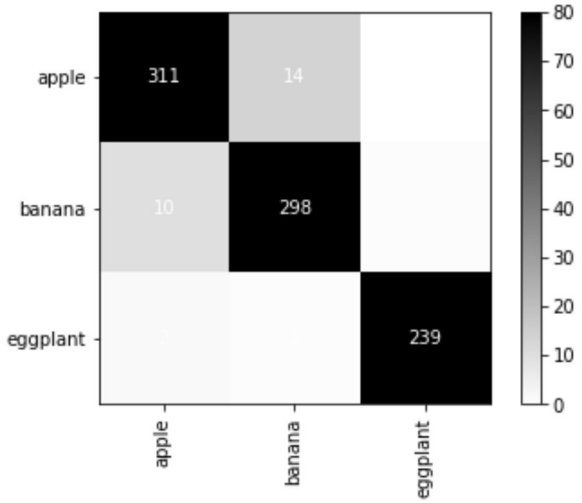

```
[[307  11   7]
 [ 20 278  11]
 [  0   2 240]]
```



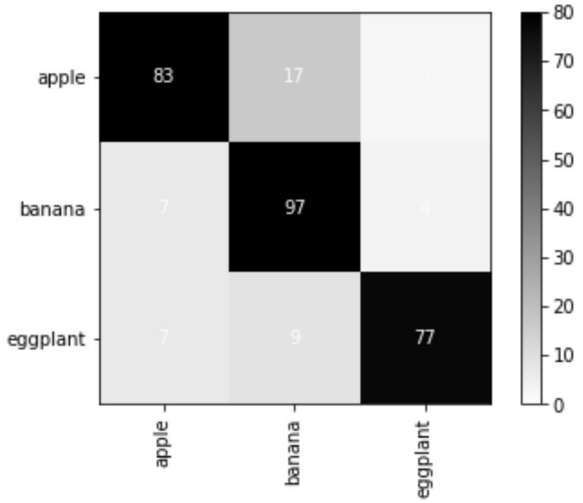
```
[[77 1   7]
 [12 85 11]
 [   11 73]]
```



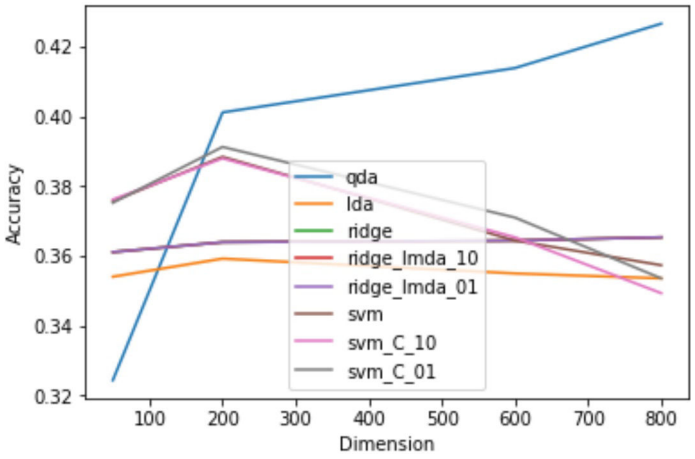

```
[[311  14   0]
 [ 10 2 8   1]
 [  2   1 23 ]]
```

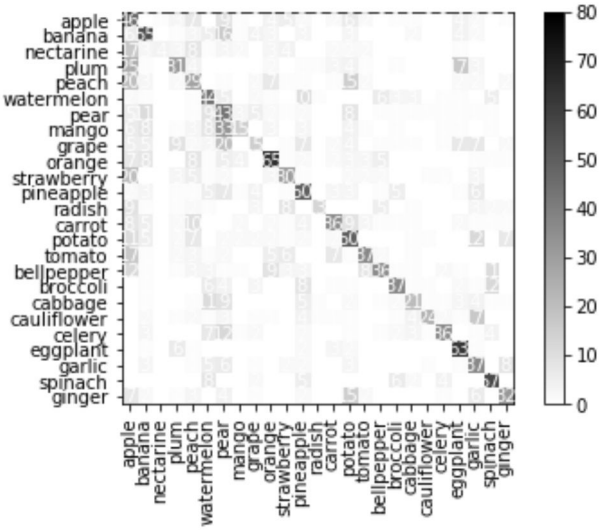



```
[[83 17  3]
 [ 7  7  4]
 [ 7    77]]
```


apple
banana
eggplant
83
17
3
7
97
4
7
9
77
80
70
60
50
40
30
20
10
0
apple
banana
eggplant

In [8]:
```
# hyper_search.py

import IPython
from numpy.random import uniform
import random
import time

import glob
import os
import pickle
import matplotlib.pyplot as plt

import numpy as np
import numpy.linalg as LA

import sys


from projection import Project2D, Projections

from confusion_mat import getConfusionMatrix
from confusion_mat import plotConfusionMatrix

from ridge_model import Ridge_Model
from qda_model import QDA_Model
from lda_model import LDA_Model
from svm_model import SVM_Model


CLASS_LABELS = ['apple','banana','nectarine','plum','peach','watermelon','pear',
'mango','grape','orange','strawberry','pineapple',
        'radish','carrot','potato','tomato','bellpepper','broccoli','cabbage','c
auliflower','celery','eggplant','garlic','spinach','ginger']


def eval_model(X,Y,k,model_key,proj):
        # PROJECT DATA
        cca_proj,white_cov = proj.cca_projection(X,Y,k=k)

        X_p = proj.project(cca_proj,white_cov,X)
        X_val_p = proj.project(cca_proj,white_cov,X_val)

        # TRAIN MODEL
        model = models[model_key]

        model.train_model(X_p,Y)
        acc,cm = model.test_model(X_val_p,Y_val)

        return acc,cm


class Model():
        """ Generic wrapper for specific model instance. """


        def __init__(self,model):
                """ Store specific pre-initialized model instance. """

                self.model = model


        def train_model(self,X,Y):
```


0.42
0.40
0.38
0.36
0.34
0.32
Accuracy
100
200
300
400
500
600
700
800
Dimension
qda
lda
ridge
ridge_lmda_10
ridge_lmda_01
svm
svm_C_10
svm_C_01


apple
banana
nectarine
plum
peach
watermelon
pear
mango
grape
orange
strawberry
pineapple
radish
carrot
potato
tomato
bellpepper
broccoli
cabbage
cauliflower
celery
eggplant
garlic
spinach
ginger
80
70
60
50
40
30
20
10
0

```
from numpy.random import uniform
from numpy.random import randn
import random
import time

import matplotlib.pyplot as plt

from scipy.linalg import eig
from scipy.linalg import sqrtm
from numpy.linalg import inv
from numpy.linalg import svd

from utils import create_one_hot_label
from utils import subtract_mean_from_data
from utils import compute_covariance_matrix

import numpy as np
import numpy.linalg as LA

import sys
from numpy.linalg import svd


class Project2D():

    '''
    Class to draw projection on 2D scatter space
    '''

    def __init__(self,projection, clss_labels):

        self.proj = projection
        self.clss_labels = clss_labels


    def project_data(self,X,Y,white=None):

        '''
        Takes list of state space and class labels
        State space should be 2D
        Labels shoud be int
        '''

        p_a = []
        p_b = []
        p_c = []

        # Project all Data
        proj = np.matmul(self.proj,white)

        X_P = np.matmul(proj,np.array(X).T)

        for i in range(len(Y)):

            if Y[i] == 0:
                p_a.append(X_P[:,i])
            elif Y[i] == 1:
```

```
                p_b.append(X_P[:,i])
            else:
                p_c.append(X_P[:,i])

        p_a = np.array(p_a)
        p_b = np.array(p_b)
        p_c = np.array(p_c)

        plt.scatter(p_a[:,0],p_a[:,1],label = 'apple')
        plt.scatter(p_b[:,0],p_b[:,1],label = 'banana')
        plt.scatter(p_c[:,0],p_c[:,1],label = 'eggplant')

        plt.legend()

        plt.show()


class Projections():

    def __init__(self,dim_x,classes):

        '''
        dim_x: the dimension of the state space x
        classes: The list of class labels
        '''

        self.d_x = dim_x
        self.NUM_CLASSES = len(classes)


    def get_random_proj(self):
        '''
        Return A which is size 2 by 729
        '''
        A = np.zeros( (2, 729), "float" )
        for i in range(A.shape[0]):
            for j in range(A.shape[1]):
                A[i, j] = random.gauss(0, 1)
        return A


    def pca_projection(self,X,Y):

        '''
        Return U_2^T
        '''
        X_demeaned, _ = subtract_mean_from_data(X, X)
        Sigma_XX = compute_covariance_matrix(X_demeaned, X_demeaned)
        U, L, V = svd(Sigma_XX)
        U2 = U[:, [0, 1]]
        return U2.T


    def cca_projection(self,X,Y,k=2):

        '''
        Return U_K^T, \Simgma_{XX}^{-1/2}
```

```
115             '''
116
117             ###SCALE AN IDENTITY MATRIX BY THIS TERM AND ADD TO COMPUTED COVARIANCE
                MATRIX TO PREVENT IT BEING SINGULAR ###
118             reg = 1e-5
119
120             y_one_hot = create_one_hot_label(Y, self.NUM_CLASSES)
121             X, y_one_hot = subtract_mean_from_data(X, y_one_hot)
122             Sigma_XX = compute_covariance_matrix(X, X)
123             Sigma_XX += reg * np.eye(Sigma_XX.shape[0])
124             Sigma_XY = compute_covariance_matrix(X, y_one_hot)
125             Sigma_YY = compute_covariance_matrix(y_one_hot, y_one_hot)
126             Sigma_YY += reg * np.eye(Sigma_YY.shape[0])
127             Sigma_XX_inv_sqrt = inv(sqrtm(Sigma_XX))
128             triple_Sigma = Sigma_XX_inv_sqrt.dot(Sigma_XY).dot( inv(sqrtm(Sigma_YY)) )
129             U, L, V = svd(triple_Sigma)
130             Uk = U[:, list(range( min(k, U.shape[1]) ))]
131             return Uk.T, Sigma_XX_inv_sqrt
132
133
134         def project(self,proj,white,X):
135             '''
136             proj, numpy matrix to perform projection
137             whit, numpy matrix to perform whitenting
138             X, list of states
139             '''
140
141             proj = np.matmul(proj,white)
142
143             X_P = np.matmul(proj,np.array(X).T)
144
145             return list(X_P.T)
146
147
148     if __name__ == "__main__":
149
150         X = list(np.load('little_x_train.npy'))
151         Y = list(np.load('little_y_train.npy'))
152
153         CLASS_LABELS = ['apple','banana','eggplant']
154
155         feat_dim = max(X[0].shape)
156         projections = Projections(feat_dim,CLASS_LABELS)
157
158         rand_proj = projections.get_random_proj()
159         # Show Random 2D Projection
160         proj2D_viz = Project2D(rand_proj,CLASS_LABELS)
161         proj2D_viz.project_data(X,Y, white = np.eye(feat_dim))
162
163         #PCA Projection
164         pca_proj = projections.pca_projection(X,Y)
165         #Show PCA 2D Projection
166         proj2D_viz = Project2D(pca_proj,CLASS_LABELS)
167         proj2D_viz.project_data(X,Y, white = np.eye(feat_dim))
168
169         #CCA Projection
170         cca_proj,white_cov = projections.cca_projection(X,Y)
```

```
171         #Show CCA 2D Projection
172         proj2D_viz = Project2D(cca_proj,CLASS_LABELS)
173         proj2D_viz.project_data(X,Y,white = white_cov)
174 
175 
176     ##############################
177 
178 
179     from numpy.random import uniform
180     import random
181     import time
182 
183     import numpy as np
184     import numpy.linalg as LA
185 
186     import sys
187 
188     from sklearn.linear_model import Ridge
189 
190     from utils import create_one_hot_label
191 
192 
193     class Ridge_Model():
194 
195         def __init__(self,class_labels):
196 
197             ###RIDGE HYPERPARAMETER
198             self.lmda = 1.0
199             self.class_labels = class_labels
200             self.ridge_model = Ridge(self.lmda)
201 
202 
203         def train_model(self,X,Y):
204             ''''
205             FILL IN CODE TO TRAIN MODEL
206             MAKE SURE TO ADD HYPERPARAMTER TO MODEL
207 
208             '''
209 
210             X = np.array(X)
211             y_one_hot = create_one_hot_label(Y, len(self.class_labels))
212             self.ridge_model.fit(X, y_one_hot)
213 
214 
215         def eval(self,x):
216             ''''
217             Fill in code to evaluate model and return a prediction
218             Prediction should be an integer specifying a class
219             '''
220             x = x.reshape(1, -1)
221             y = self.ridge_model.predict(x)
222             return np.argmax(y)
223 
224 
225     ##########################
226 
227 
```

```
228 import random
229 import time
230
231 import glob
232 import os
233 import pickle
234 import matplotlib.pyplot as plt
235
236 import numpy as np
237 import numpy.linalg as LA
238
239 import sys
240 from numpy.linalg import inv
241 from numpy.linalg import det
242 from sklearn.svm import LinearSVC
243 from projection import Project2D, Projections
244
245 from utils import subtract_mean_from_data
246 from utils import compute_covariance_matrix
247
248
249 class LDA_Model():
250
251     def __init__(self,class_labels):
252
253         ###SCALE AN IDENTITY MATRIX BY THIS TERM AND ADD TO COMPUTED COVARIANCE
          MATRIX TO PREVENT IT BEING SINGULAR ###
254         self.reg_cov = 0.001
255         self.NUM_CLASSES = len(class_labels)
256
257
258     def train_model(self,X,Y):
259         ''''
260         FILL IN CODE TO TRAIN MODEL
261         MAKE SURE TO ADD HYPERPARAMTER TO MODEL
262
263         '''
264         ps = [ [] for j in range(self.NUM_CLASSES) ]
265         for i, y in enumerate(Y):
266             ps[y].append(X[i])
267
268         self.mean_list = []
269         for lst in ps:
270             self.mean_list.append( np.mean(np.array(lst), axis=0) )
271
272         Sigma_XX = compute_covariance_matrix(X, X)
273         Sigma_XX += self.reg_cov * np.identity(Sigma_XX.shape[0])
274         self.Sigma_inv = inv(Sigma_XX)
275
276
277     def eval(self,x):
278         ''''
279         Fill in code to evaluate model and return a prediction
280         Prediction should be an integer specifying a class
281         '''
282         x = x.reshape(1, -1)
283         y = {}
```

```
            for i in range(self.NUM_CLASSES):
                x_demeaned = x - self.mean_list[i]
                f = - x_demeaned.dot(self.Sigma_inv).dot(x_demeaned.T)
                y[i] = f.flatten()[0]
            return max(y, key=lambda x: y[x])


############################


import random
import time

import numpy as np
import numpy.linalg as LA

from numpy.linalg import inv
from numpy.linalg import det

from projection import Project2D, Projections

from utils import subtract_mean_from_data
from utils import compute_covariance_matrix


class QDA_Model():

    def __init__(self,class_labels):

        ###SCALE AN IDENTITY MATRIX BY THIS TERM AND ADD TO COMPUTED COVARIANCE MATRIX TO PREVENT IT BEING SINGULAR ###
        self.reg_cov = 0.01
        self.NUM_CLASSES = len(class_labels)


    def train_model(self,X,Y):
        ''''
        FILL IN CODE TO TRAIN MODEL
        MAKE SURE TO ADD HYPERPARAMTER TO MODEL

        '''
        ps = [ [] for j in range(self.NUM_CLASSES) ]
        for i, y in enumerate(Y):
            ps[y].append(X[i])

        self.mean_list = []
        self.Sigma_inv_list = []
        for lst in ps:
            self.mean_list.append( np.mean(np.array(lst), axis=0) )
            Sigma_XX = compute_covariance_matrix(lst, lst)
            Sigma_XX += self.reg_cov * np.identity(Sigma_XX.shape[0])
            self.Sigma_inv_list.append(inv(Sigma_XX))


    def eval(self,x):
        ''''
        Fill in code to evaluate model and return a prediction
```

```
340             Prediction should be an integer specifying a class
341             '''
342             x = x.reshape(1, -1)
343             y = {}
344             for i in range(self.NUM_CLASSES):
345                 x_demeaned = x - self.mean_list[i]
346                 f = - x_demeaned.dot(self.Sigma_inv_list[i]).dot(x_demeaned.T)
347                 y[i] = f.flatten()[0]
348             return max(y, key=lambda x: y[x])
349
350
351     ########################
352
353
354     from numpy.random import uniform
355     import random
356     import time
357
358     import matplotlib.pyplot as plt
359
360     import numpy as np
361     import numpy.linalg as LA
362
363     import sys
364
365     from sklearn.svm import LinearSVC
366     from projection import Project2D, Projections
367
368     from utils import create_one_hot_label
369
370
371     class SVM_Model():
372
373         def __init__(self,class_labels,projection=None):
374
375             ###SLACK HYPERPARAMETER
376             self.C = 1.0
377             self.class_labels = class_labels
378             self.svc_model = LinearSVC(C=self.C)
379
380
381         def train_model(self,X,Y):
382             ''''
383             FILL IN CODE TO TRAIN MODEL
384             MAKE SURE TO ADD HYPERPARAMTER TO MODEL
385
386             '''
387             X = np.array(X)
388             self.svc_model.fit(X, Y)
389
390
391         def eval(self,x):
392             ''''
393             Fill in code to evaluate model and return a prediction
394             Prediction should be an integer specifying a class
395             '''
396             x = x.reshape(1, -1)
```

```
397             y = self.svc_model.predict(x)
398             return y[0]
399
```