```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 08 17:36:34 2020

@author: zongh
"""

from sklearn.datasets import load_breast_cancer
import numpy as np
import matplotlib.pyplot as plt

# import data
breast_cancer = load_breast_cancer()
X = breast_cancer.data # dimension is 30, we need to convert it to 2
Y = breast_cancer.target
#########################################################
"""
Problem 1
"""
############
"""
Q1
"""
def pca(matrix, dimension):
    # covariance
    matrix_cov = np.cov(np.transpose(matrix))
    # SVD
    U, S, V = np.linalg.svd(matrix_cov)
    # calculate eigenvalue and eigenvector
    U_eigVal, U_eigVec = np.linalg.eig(U)
    # extrac the largest two eigenvalues' indices
    idx = U_eigVal.argsort()[-dimension:][::-1]
    # output corresponding projection matrix
    U_proj = U[:,idx]
    return U_proj

############
"""
Q2

One needs to normalize the data. Specifically, the
mean normalization, and feature scaling.
"""
############
"""
Q3
"""

def normalization(data):
    # mean normalization in each dimension across all examples
    data_mean = np.mean(data,axis=0)
    data_std = np.std(data,axis=0)
    data_norm = data - np.array(data.shape[0]*[data_mean])
```

```
    # scaling in each dimension across all examples
    data_norm = data_norm / data_std[None,:]
    return data_norm

X_norm = normalization(X)
# PCA
dim = 2
U_2d = pca(X_norm, dim)
X_2d = np.matmul(X_norm, U_2d)
# plot the data
plt.figure()
plt.plot(X_2d[:,dim-2], X_2d[:,dim-1], 'bo')
"""
The coordinate of each point is a projection of the original
data that is in the dimension of 30, to the dimension of 2.
It is a 2D representation of the original 30D representation.
"""

############################################################
"""
Problem 2
"""
############
"""
Q1
"""
# initilize the centroid position , randomly select k point from training data
# as cluster centroids

# k-mean clustering
def k_mean(k, data):
    distance = np.zeros((data.shape[0],k))
    centroid = np.zeros((k,data.shape[1]))
    distortion = 0
    for i in range(k):
        # initialization by randomly selecting training data as centroids
        centroid[i] = data[np.random.randint(0, data.shape[0]-1)]

    for i in range(k):
        # assignment
        distance[:,i] = np.sqrt((data[:,0]-centroid[i,0])**2 + (data[:,1]-
centroid[i,1])**2)
        assignment = np.argmin(distance,axis=1)

    for i in range(k):
        # update centroid
        centroid[i][0] = np.mean(data[np.where(assignment == i)][:,0])
        centroid[i][1] = np.mean(data[np.where(assignment == i)][:,1])

    for i in range(k):
        # calculate distortion
        distortion = distortion + np.sum((centroid[i][0] -
data[np.where(assignment == i)][:,0])**2 +
```

The coordinate of each point re

```
                        (centroid[i][1] - data[np.where(assignment ==
i)][:,1])**2)
    distortion = distortion/data.shape[0]

    return centroid, assignment, distortion

###########
"""
Q2
To calculate the distortion, the input will be the L2 distance between the data
and their labelled centroids.
"""
# see the distortion
dist = np.zeros((6,1))
for i in range(2,8):
    _, _, distortion = k_mean(i, X_2d);
    dist[i-2,:] = distortion
    print(distortion)

# plot the distirtion w.r.t. k
plt.figure()
plt.plot([2,3,4,5,6,7], dist, 'bo')

###########
"""
Q3
The plot varies from run-to-run due to the random initializations of the
centroids, the pivot point is around k = 4-5, but in general the more centroids
one has, the less distortion one gets. One can pick k=7 and achieve in general
the smallest distortion compared to others from 2 to 7.
"""
###########
"""
Q4
"""
# we choose k = 7
k = 7
centroid, assignment, _ = k_mean(k, X_2d);
###########
"""
Q5
"""
###########
plt.figure()
plt.plot(X_2d[:,dim-2], X_2d[:,dim-1], 'bo')

# we have 7 clusters so 7 colors
plt.plot(centroid[0,dim-2], centroid[0,dim-1], 'rx')
plt.plot(X_2d[np.where(assignment == 0)][:,dim-2], X_2d[np.where(assignment ==
0)][:,dim-1], 'ro')

plt.plot(centroid[1,dim-2], centroid[1,dim-1], 'gx')
```

```
plt.plot(X_2d[np.where(assignment == 1)][:,dim-2], X_2d[np.where(assignment ==
1)][:,dim-1], 'go')

plt.plot(centroid[2,dim-2], centroid[2,dim-1], 'yx')
plt.plot(X_2d[np.where(assignment == 2)][:,dim-2], X_2d[np.where(assignment ==
2)][:,dim-1], 'yo')

plt.plot(centroid[3,dim-2], centroid[3,dim-1], 'wx')
plt.plot(X_2d[np.where(assignment == 3)][:,dim-2], X_2d[np.where(assignment ==
3)][:,dim-1], 'wo')

plt.plot(centroid[4,dim-2], centroid[4,dim-1], 'mx')
plt.plot(X_2d[np.where(assignment == 4)][:,dim-2], X_2d[np.where(assignment ==
4)][:,dim-1], 'mo')

plt.plot(centroid[5,dim-2], centroid[5,dim-1], 'kx')
plt.plot(X_2d[np.where(assignment == 5)][:,dim-2], X_2d[np.where(assignment ==
5)][:,dim-1], 'ko')

plt.plot(centroid[6,dim-2], centroid[6,dim-1], 'cx')
plt.plot(X_2d[np.where(assignment == 6)][:,dim-2], X_2d[np.where(assignment ==
6)][:,dim-1], 'co')

##########################################################
"""
Problem 3
"""
############
"""
Q1
We can partition the dataset like we did in assignment 2, where the ratio between
training data and test data is close 4:1.
"""
train_X = X[:450]
test_X = X[450:]
train_Y = Y[:450]
test_Y = Y[450:]
############
"""
Q2
Validation sets are used to find hyperparameters. In linear classification
situation we do not have any hyperparameters here, therefore we do not need
validation data.
"""
############
"""
Q3
See the proof later.
"""
############
"""
Q4
"""
```

```
weight =
np.matmul(np.matmul(np.linalg.inv(np.matmul(np.transpose(train_X),train_X)),
                  np.transpose(train_X)), train_Y)

"""
weight:
0.340363
-0.01843
-0.0117673
-0.00181353
-0.618911
-0.292204
-0.71161
-3.15772
1.33157
28.1621
-0.266302
-0.0403557
0.000766271
0.00132855
-12.7391
3.82734
3.38385
-9.86539
0.391343
-20.0373
-0.238546
0.00381197
0.00406451
0.00124569
0.263685
0.0424862
-0.486999
-0.886331
-0.866581
-5.02111
"""
###########
"""
Q5
"""
z = np.matmul(test_X, weight)
# prediction
y = 1/(1+np.exp(-z))

for i in range(y.shape[0]):
    if y[i] > 0.5:
        y[i] = 1
    else:
        y[i] = 0

p = np.shape(np.nonzero(y - test_Y))[1]
precision = 1 - float(p)/y.shape[0]
```

```
# precision is about 84%
###########
"""
Q6
"""
z_tot = np.matmul(X, weight)
y_tot = 1/(1+np.exp(-z_tot))

for i in range(y_tot.shape[0]):
    if y_tot[i] > 0.5:
        y_tot[i] = 1
    else:
        y_tot[i] = 0

p_tot = np.shape(np.nonzero(y_tot - Y))[1]
precision_tot = 1 - float(p_tot)/y_tot.shape[0]
# after applying w to all data, precision is about 75%
plt.figure()
plt.plot(X_2d[:,dim-2], X_2d[:,dim-1], 'bo')
plt.plot(X_2d[np.where(y_tot == 0)][:,dim-2], X_2d[np.where(y_tot == 0)][:,dim-
1], 'ro')
###########
```

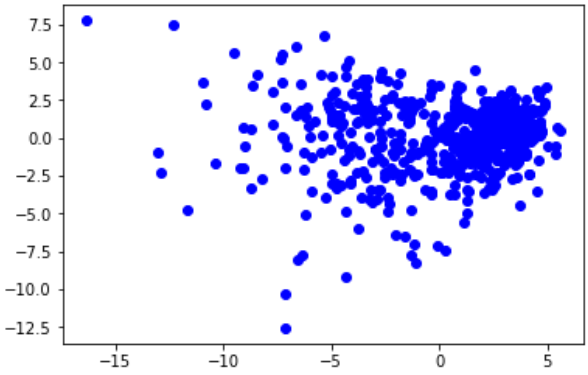

Figure 1 Problem 1 Q1

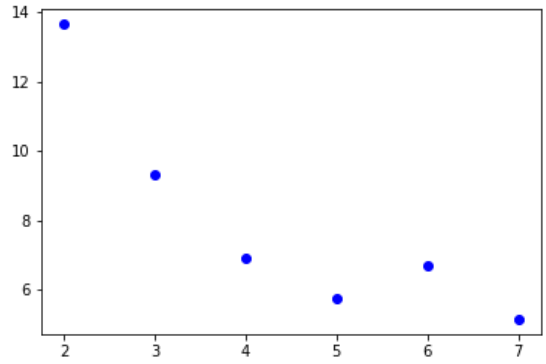

Figure 2 Problem 2 Q2

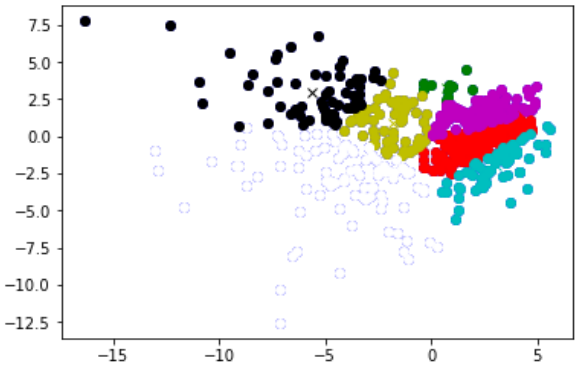

Figure 3 Problem 2 Q5

Missing the centroids on th

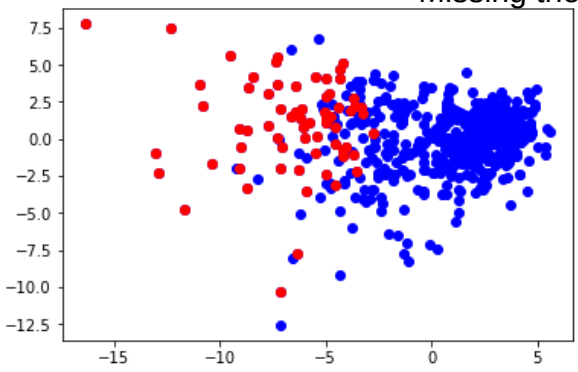

Figure 4 Problem 3 Q6, ref points are sorted as "0"

Problem 3

3.

We model this as a least-square problem in general

$$XW = Y$$

We see here for this question it is an overdetermined system since there are more input points (569) than the dimension (30). Therefore, strictly speaking there is no X that really satisfied the above equation. We can only do the following:

$$\min_W ||XW - Y||_2^2$$

Let

$$f(W) = ||XW - Y||_2^2 = (XW - Y)^T(XW - Y) = W^T X^T XW - 2Y^T XW + Y^T Y$$

$$\nabla_W f(W) = 2X^T XW - 2X^T Y = 0 \rightarrow W^* = (X^T X)^{-1} X^T Y$$

Problem 4

1. Lets assume $\epsilon = 0$ for now, since

$$Var[g] = E[g^2] - E[g]^2 \geq 0 \rightarrow \sqrt{E[g^2]} \geq |E[g]| \rightarrow \frac{|E[g]|}{\sqrt{E[g^2]}} \leq 1$$

$$\Delta w = \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t}} = \alpha \left| \frac{E[g]}{\sqrt{E[g^2]}} \right| \leq \alpha$$

Where g is the gradient. Therefore, the learning rate sets up an upper bound for the step size. Therefore, we can relax the right order of α compared to other update rules. Therefore, it becomes not very important to fine tune the learning rate for Adam. incorrect

2. Since we have only one GPU, it is appropriate to use Mini Batch Gradient Decent to train the network. Specific to this variant, we will pick a batch size that contains some portion of the examples that gives the optimal result.

3. Due to the access of multiple GPU, it is better to process data in parallel, we will still use Mini Batch Gradient Decent to train the network also due to the convexity concern. Again, we will pick a batch size that contains some portion of the examples that gives the optimal result. One batch will be split evenly and randomly across all GPUs. The gradient will be computed locally in each GPU, then the gradients from each GPU will be summed before updating the weights. Therefore, we need to wait until all gradient descents are completed across all GPUs before the next epoch. In this case the data is parallelly fed to all GPUs. -1, batch size sele

Problem 5

1. It seems one want to search for the optimized hyperparameters here, but there are a few problems
   - The weight update rule in gradient descent is defined as $w = \mu w - \alpha \frac{\partial L}{\partial w}$ where μ is the momentum. If the momentum is 0 then the weight will not be updated, unless $w = (1 - \mu)w - \alpha \frac{\partial L}{\partial w}$
   - A big concern is the generalization after splitting the training/test data further. Since in each step, the obtained optimized hyperparameters are only "optimized" to the smaller subsets of the samples. Therefore, the hypermeters may only "overfit" to each subset and "underfit" to other subset.

- To obtain a better generalization, one should train the network using all training sets. The hyperparameters should be tuned by using validation set rather than test set.

2. First, a confusion matrix A should be a square matrix, and its diagonal represents the corrected classified number of test examples. The off-diagonal entries in each column (class) are the wrong classified samples. Confusion matrix is preferable when there are multiple classes and sample size for each class is different from each other. -1, justification is missing
3. The bias of the model:

$$bias = (E[y] - E[t])^2$$

where y is the predicted result and t is the actual label.
The bias of the model:

$$variance = Var[y]$$

These two lines will be calculated each time after one epoch. In each epoch, the bias and variance will be calculated and monitor and if they both decrease with respect to time, then it is a good algorithm. If the bias going higher and variance goes lower with respect to each epoch, then likely the system is underfitting; if the bias going lower and variance goes higher with respect to each epoch, then likely the system is overfitting,