

ELL-409 (Machine Intelligence and Learning)

Assignment 2

Report

Submitted by – Mihir Gupta , 2018PH10816

1) Principal Component Regression

In statistics, principal component regression (PCR) is a regression analysis technique that is based on principal component analysis (PCA). More specifically, PCR is used for estimating the unknown regression coefficients in a standard linear regression model.

In PCR, instead of regressing the dependent variable on the explanatory variables directly, the principal components of the explanatory variables are used as regressors. One typically uses only a subset of all the principal components for regression, making PCR a kind of regularized procedure and also a type of shrinkage estimator.

Often the principal components with higher variances (the ones based on eigenvectors corresponding to the higher eigenvalues of the sample variance-covariance matrix of the explanatory variables) are selected as regressors. However, for the purpose of predicting the outcome, the principal components with low variances may also be important, in some cases even more important.^[1]

One major use of PCR lies in overcoming the multicollinearity problem which arises when two or more of the explanatory variables are close to being collinear.^[2] PCR can aptly deal with such situations by excluding some of the low-variance principal components in the regression step. In addition, by usually regressing on only a subset of all the principal components, PCR can result in dimension reduction through substantially lowering the effective number of parameters characterizing the underlying model. This can be particularly useful in settings with high-dimensional covariates. Also, through appropriate selection of the principal components to be used for regression, PCR can lead to efficient prediction of the outcome based on the assumed model.

a) Dataset used – Breast cancer dataset

```
5  
6 df = pd.read_csv('./breastcancer.csv')  
7  
8  
9  
10  
11 df  
12
```

Out[1]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...	te
0	842302	M	17.99	10.38	122.80	1001.0	0.11640	0.27760	0.30010	0.14710	...	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	...	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	...	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	...	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	...	
...
564	926424	M	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	...	
565	926682	M	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	...	
566	926954	M	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	...	
567	927241	M	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	...	
568	92751	B	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	...	

69 rows × 33 columns

b) Generate co-variance matrix

```

In [3]: # PCA to reduce dataset features to 2

from sklearn.preprocessing import StandardScaler
stdX = StandardScaler().fit_transform(X)

import numpy as np
means = np.mean(stdX, axis=0)
covariances = (stdX - means).T.dot((stdX - means)) / (stdX.shape[0]-1)
print('Covariance matrix \n%s' %covariances)

print('NumPy covariance matrix: \n%s' %np.cov(stdX.T))

covariances = np.cov(stdX.T)

eigenvals, eigenvecs = np.linalg.eig(covariances)

print('Eigenvectors \n%s' %eigenvecs)
print('\nEigenvalues \n%s' %eigenvals)

cmat = np.corrcoef(X.T)

eigenvals, eigenvecs = np.linalg.eig(cmat)

print('Eigenvectors \n%s' %eigenvecs)
print('\nEigenvalues \n%s' %eigenvals)

```

Covariance matrix

```

[[ 1.00176056e+00  3.24351929e-01  9.99612069e-01  9.89095475e-01
   1.70881506e-01  5.07014640e-01  6.77955036e-01  8.23976636e-01
   1.48001350e-01 -3.12179472e-01  6.80285970e-01 -9.74887767e-02
   6.75358538e-01  7.37159198e-01 -2.22992026e-01  2.06362656e-01
   1.94545531e-01  3.76831225e-01 -1.04504545e-01 -4.27163418e-02
   9.71245907e-01  2.97530545e-01  9.66835698e-01  9.42739295e-01
   1.19826732e-01  4.14190751e-01  5.27839123e-01  7.45524434e-01
   1.64241985e-01  7.07832563e-03]

[ 3.24351929e-01  1.00176056e+00  3.30113223e-01  3.21650988e-01
 -2.34296930e-02  2.37118951e-01  3.02950254e-01  2.93980713e-01
  7.15266864e-02 -7.65717560e-02  2.76354360e-01  3.87037830e-01

```

c) Determine eigenvectors and eigenvalues and print in descending order

Eigenvalues

```
[1.32816077e+01 5.69135461e+00 2.81794898e+00 1.98064047e+00  
1.64873055e+00 1.20735661e+00 6.75220114e-01 4.76617140e-01  
4.16894812e-01 3.50693457e-01 2.93915696e-01 2.61161370e-01  
2.41357496e-01 1.57009724e-01 9.41349650e-02 7.98628010e-02  
5.93990378e-02 5.26187835e-02 4.94775918e-02 1.33044823e-04  
7.48803097e-04 1.58933787e-03 6.90046388e-03 8.17763986e-03  
1.54812714e-02 1.80550070e-02 2.43408378e-02 2.74394025e-02  
3.11594025e-02 2.99728939e-02]
```

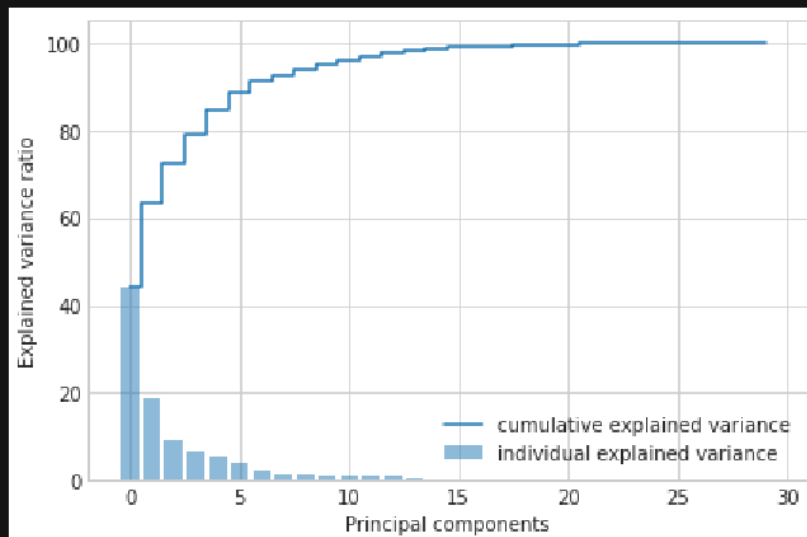
d) plot cumulative / individual variance of principal components

```
[ 0.22876753  0.09796411]
```

```
[ 0.25088597 -0.00825724]
```

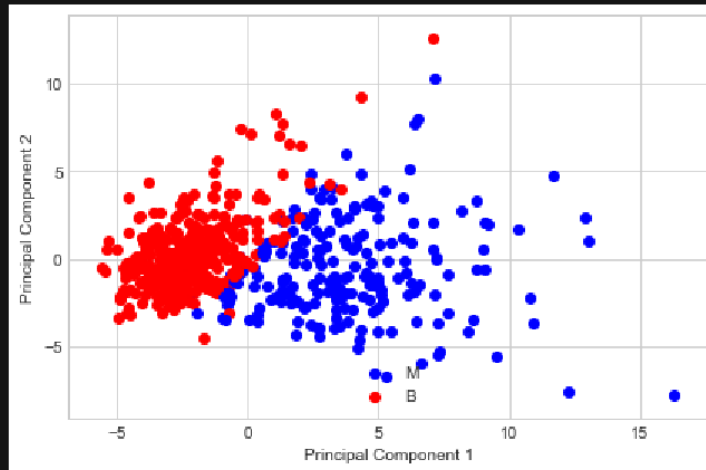
```
[ 0.12290456  0.14188335]
```

```
[ 0.13178394  0.27533947]]
```



e) Print scatter plot of principal components

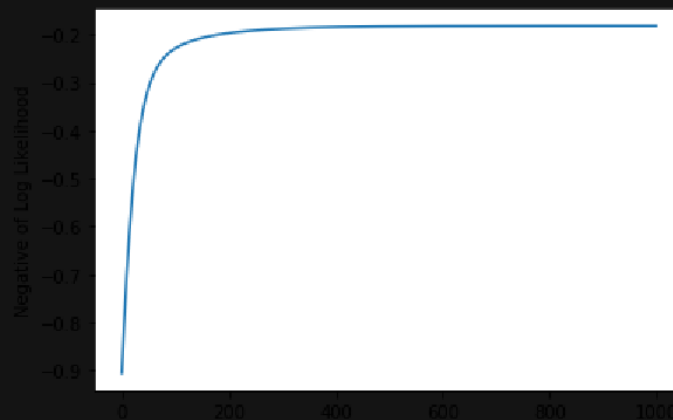
```
1.08425474e+00 1.22168435e-01 -2.13559838e-01 1.29284790e+00
-1.78019691e-01 -2.13110611e+00 3.48263666e+00 -5.73854672e-01
-3.57681744e+00 -3.58404786e+00 -1.90229671e+00 1.67201011e+00
-6.70636791e-01]
```



f) Perform PCR analysis and plot regression analysis

Negative of loss function value vs No. of iterations

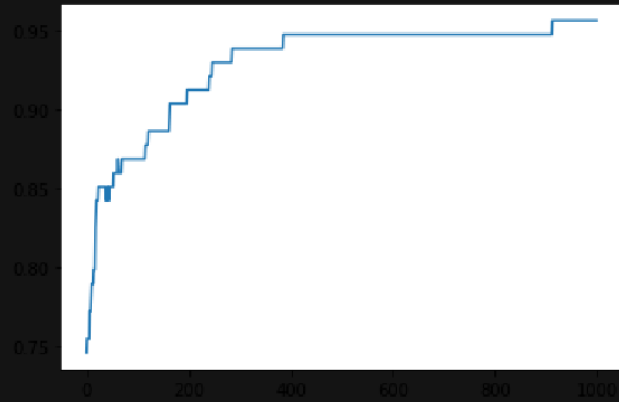
```
In [9]: plt.plot(loss)
plt.ylabel("Negative of Log Likelihood")
plt.xlabel("Time")
plt.show()
```



Accuracy on test data vs No. of iterations

```
In [10]: # Accuracy with iterations
```

```
plt.plot(acc)  
plt.show()  
print(acc[-1])
```

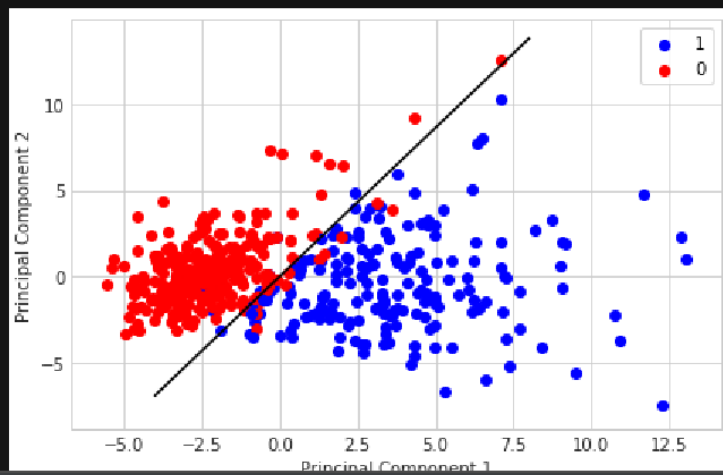


0.956140350877193

```
x = np.linspace(-4,8,10)  
y = -(w[0]*x + b)/w[1]  
plt.plot(x,y,color='k')
```

```
plt.legend()  
plt.show()
```

455



2) Multinomial Logistic Regression

Multinomial logistic regression is a classification method that generalizes logistic regression to multiclass problems, i.e. with more than two possible discrete outcomes.^[1] That is, it is a model that is used to predict the probabilities of the different possible outcomes of a categorically distributed dependent variable, given a set of independent variables (which may be real-valued, binary-valued, categorical-valued, etc.).

Multinomial logistic regression is used when the dependent variable in question is nominal (equivalently categorical, meaning that it falls into any one of a set of categories that cannot be ordered in any meaningful way) and for which there are more than two categories.

Some examples would be:

- Which major will a college student choose, given their grades, stated likes and dislikes, etc.?
- Which blood type does a person have, given the results of various diagnostic tests?

In the binary classification, logistic regression determines the probability of an object to belong to one class among the two classes.

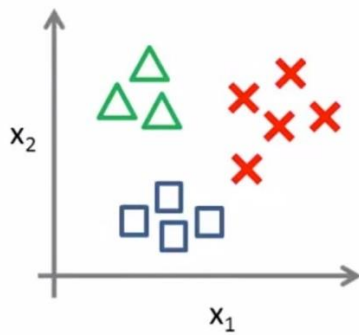
If the predicted probability is greater than 0.5 then it belongs to a class that is represented by 1 else it belongs to the class represented by 0. In multinomial logistic regression, we use the concept of one vs rest classification using binary classification technique of logistic regression.


Now, for example, let us have “K” classes. First, we divide the classes into two parts, “1” represents the 1st class and “0” represents the rest of the classes, then we apply binary classification in this 2 class and determine the probability of the object to belong in 1st class vs rest of the classes.


Similarly, we apply this technique for the “k” number of classes and return the class with the highest probability. By, this way we determine in which class the object belongs. In this way multinomial logistic regression works. Below there are some diagrammatic representation of one vs rest classification:-


Step 1:-

One-vs-all (one-vs-rest):



Class 1: 

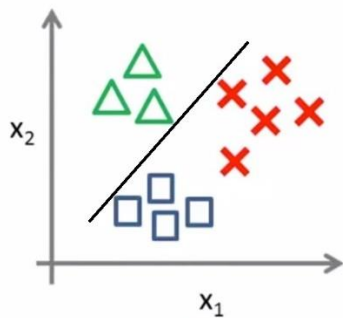
Class 2: 


Class 3: 


Here there are 3 classes represented by triangles, circles, and squares.


Step 2:

One-vs-all (one-vs-rest):



Class 1: 

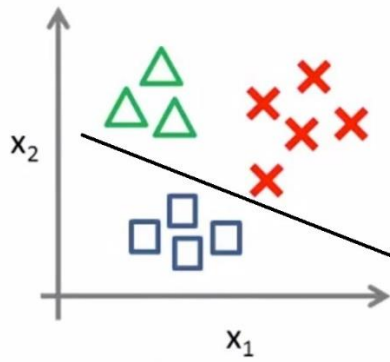
Class 2: 


Class 3: 


Here we use the one vs rest classification for class 1 and separates class 1 from the rest of the classes.


Step 3:

One-vs-all (one-vs-rest):



Class 1: 

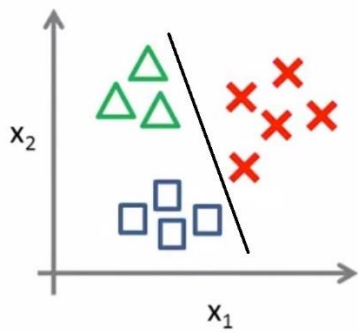
Class 2: 


Class 3: 

Here we use the one vs rest classification for class 2 and separates class 2 from the rest of the classes.


Step 4:

One-vs-all (one-vs-rest):



Class 1: 

Class 2: 

Class 3: 

a) Dataset Used – Wines

```
In [1]: import pandas as pd

df = pd.read_csv('./Winerandom.csv')
df.sample(frac=1)
print(df)
print(df.iloc[0].values)
df.tail()

2 12.29 1.61 2.21 20.4 103 1.1 1.02 0.37 1.46 3.05 0.906 \

0 1 14.23 1.71 2.43 15.6 127 2.80 3.06 0.28 2.29 5.64 1.04

1 1 13.41 3.84 2.12 18.8 90 2.45 2.68 0.27 1.48 4.28 0.91
```

b) Perform PCA

```
In [4]: # PCA

from sklearn.preprocessing import StandardScaler
stdX = StandardScaler().fit_transform(X)

import numpy as np
means = np.mean(stdX, axis=0)
covariances = (stdX - means).T.dot((stdX - means)) / (stdX.shape[0]-1)
print('Covariance matrix \n%s' %covariances)

print('NumPy covariance matrix: \n%s' %np.cov(stdX.T))
=
covariances = np.cov(stdX.T)

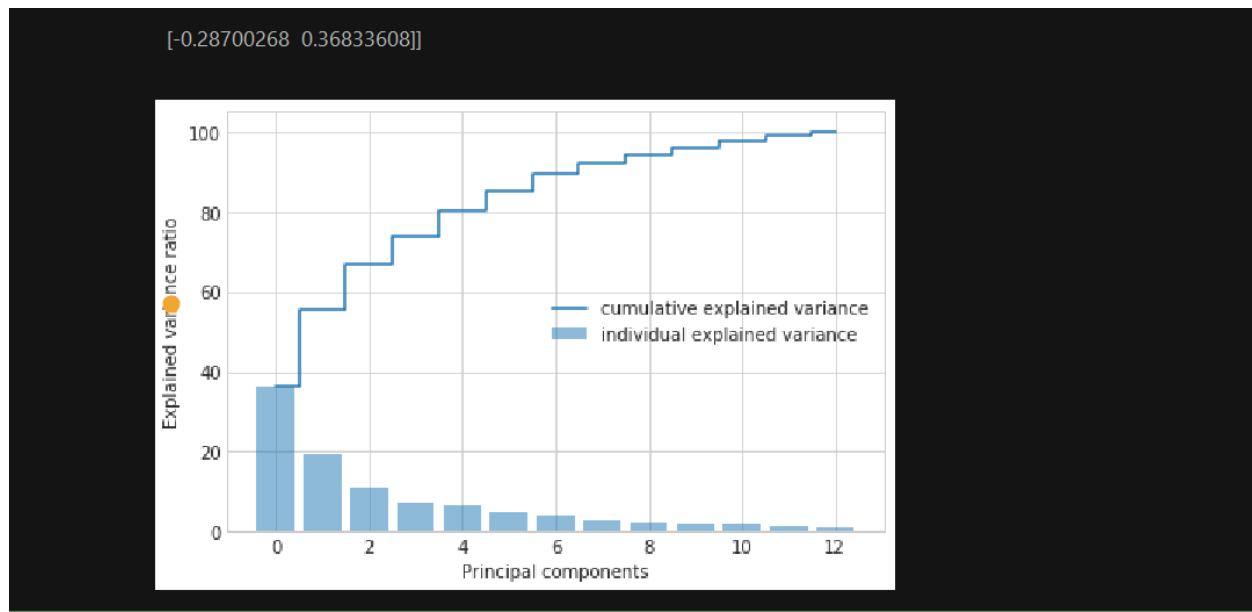
eigenvals, eigenvecs = np.linalg.eig(covariances)

print('Eigenvectors \n%s' %eigenvecs)
print('\nEigenvalues \n%s' %eigenvals)

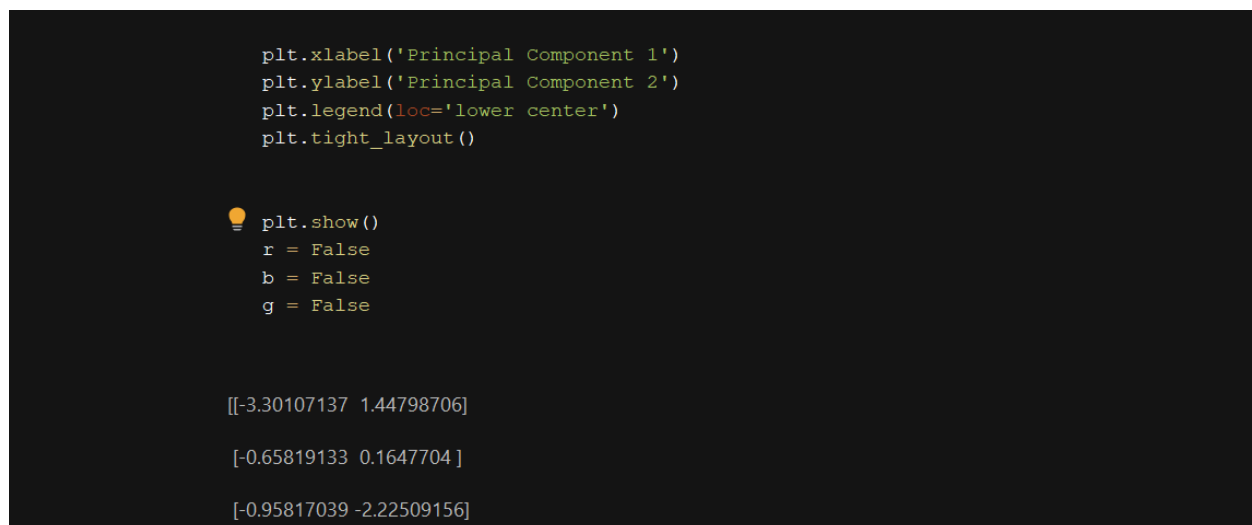
cmat = np.corrcoef(X.T)

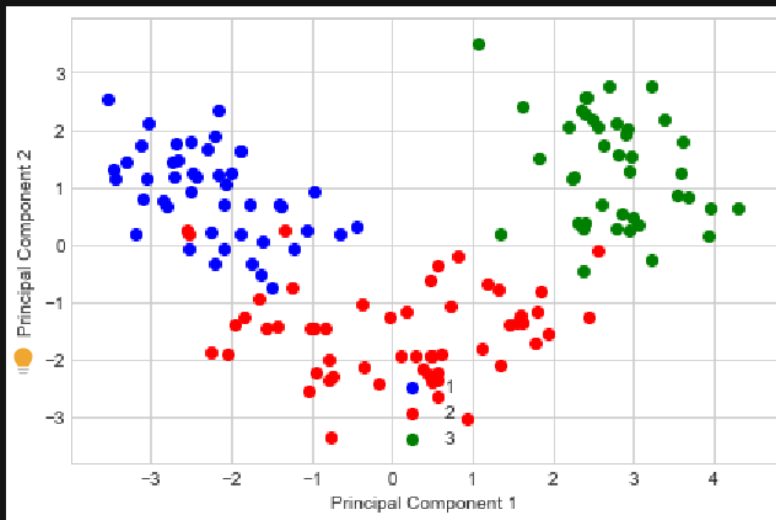
eigenvals, eigenvecs = np.linalg.eig(cmat)
```

c) Plot individual/cumulative variance of principal components



d) Create scatter plot with principal components





e) Perform multinomial regression analysis

```
In [6]: # Multinomial Regression

def hypothesis(x,w,b):
    """accepts input vector x, input weight vector w and bias b"""
    hx = np.dot(x,w)+b
    return sigmoid(hx)

def sigmoid(h):
    return 1.0/(1.0 + np.exp(-1.0*h))

def error(y,x,w,b):
    m = x.shape[0]
    err = 0.0
    for i in range(m):
        hx = hypothesis(x[i],w,b)
        err += y[i]*np.log2(hx)+(1-y[i])*np.log2(1-hx)
    return err/m

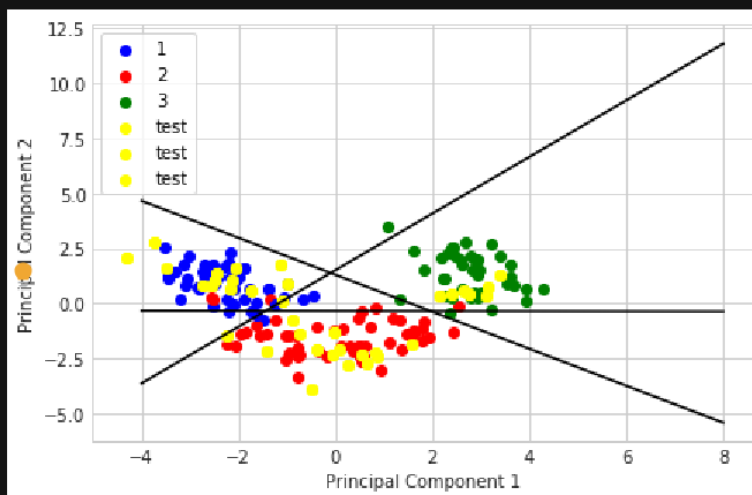
def get_grad(x,w,b,y):
    grad_b = 0.0
    grad_w = np.zeros(w.shape)
    m = x.shape[0]
    for i in range(m):
        hx = hypothesis(x[i],w,b)
        grad_w += (y[i] - hx)*x[i]
        grad_b += (y[i]-hx)
```

f) Create scatter plot with regression results

```
plt.scatter(list(X_test[:,0]), list(X_test[:,1]),label='test',c='yellow')

x = np.linspace(-4,8,10)
y = -(W[0]*x + b)/W[1]
plt.plot(x,y,color='k')

plt.legend()
plt.show()
```



g) Results accuracy – 100% for test data

```
[3 2 3 2 1 1 1 2 2 2 1 2 1 1 1 1 3 1 2 1 2 2 3 1 2 1 2 2 1 2 3 3 1 3 2 2] <class 'numpy.ndarray'>
[3 2 3 2 1 1 1 2 2 2 1 2 1 1 1 1 3 1 2 1 2 2 3 1 2 1 2 2 1 2 3 3 1 3 2 2] <class 'numpy.ndarray'>

accuracy is 100.0 %

In [9]: accuracy = (float((plabs==Y_test).sum())/s)*100
print("accuracy is ", accuracy , " %")

accuracy is 100.0 %
```

3) Mean-shift clustering

Meanshift is falling under the category of a clustering algorithm in contrast of Unsupervised learning that assigns the data points to the clusters iteratively by shifting points towards the mode (mode is the highest density of data points in the region, in the context of the Meanshift). As such, it is also known as the **Mode-seeking algorithm**. Mean-shift algorithm has applications in the field of image processing and computer vision. Unlike the popular K-Means cluster algorithm, mean-shift does not require specifying the number of clusters in advance. The number of clusters is determined by the algorithm with respect to the data

Mean shift is a procedure for locating the maxima—the **modes**—of a density function given discrete data sampled from that function.^[1] This is an iterative method, and we start with an initial estimate x . Let a **kernel function** $K(x_i - x)$ be given. This function determines the weight of nearby points for re-estimation of the mean. Typically a **Gaussian kernel** on the distance to the current estimate is used, $K(x_i - x) = e^{-c\|x_i - x\|^2}$. The weighted mean of the density in the window determined by K is

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x)x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

where $N(x)$ is the neighborhood of x , a set of points for which $K(x_i) \neq 0$.

The difference $m(x) - x$ is called **mean shift** in Fukunaga and Hostettler.^[3] The **mean-shift algorithm** now sets $x \leftarrow m(x)$, and repeats the estimation until $m(x)$ converges.

Although the mean shift algorithm has been widely used in many applications, a rigid proof for the convergence of the algorithm using a general kernel in a high dimensional space is still not known.^[4] Aliyari Ghassabeh showed the convergence of the mean shift algorithm in one-dimension with a differentiable, convex, and strictly decreasing profile function.^[5] However, the one-dimensional case has limited real world applications. Also, the convergence of the algorithm in higher dimensions with a finite number of the (or isolated) stationary points has been proved.^{[4][6]} However, sufficient conditions for a general kernel function to have finite (or isolated) stationary points have not been provided.

X-Means

The obvious shortcomings of the basic k-means clustering are that the number of clusters needs to be determined in advance and the computational cost with respect to the number of observations, clusters, and iterations.

What X-means does exactly is the following. Assuming a user supplied lower and upper bound for the number of clusters, it finds all locally optimal k-means clusterings within this range, evaluates them using the BIC criterion, and at the end returns the clustering that evaluates the best. X-means can be performed by splitting, or other random criteria. We have used mean-shift to find optimal number of clusters

a) Finding centroids

```
In [12]: # meanshift clustering

import matplotlib.pyplot as plt
import numpy as np

class Find_Centroids:
    def __init__(self, rad=2.5):
        self.rad = rad

    def fit(self, Xvec):
        centroids = {}

        for i in range(len(Xvec)):
            centroids[i] = Xvec[i]

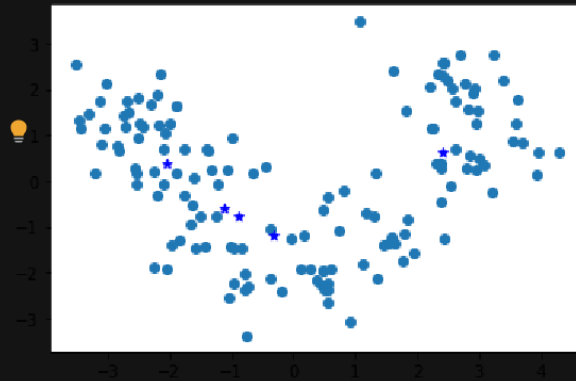
        while True:
            newcentroids = []
            for i in centroids:
                nearby = []
                centroid = centroids[i]
                for example in Xvec:
                    if np.linalg.norm(example-centroid) < self.rad:
                        nearby.append(example)
```

b) Print centroids

```
plt.scatter(X_train[:,0], X_train[:,1])

for c in centroids:
    plt.scatter(centroids[c][0], centroids[c][1], color='b', marker='*')

plt.show()
print(centroids)
print(type(centroids))
```



c) Assigning points to Centroids

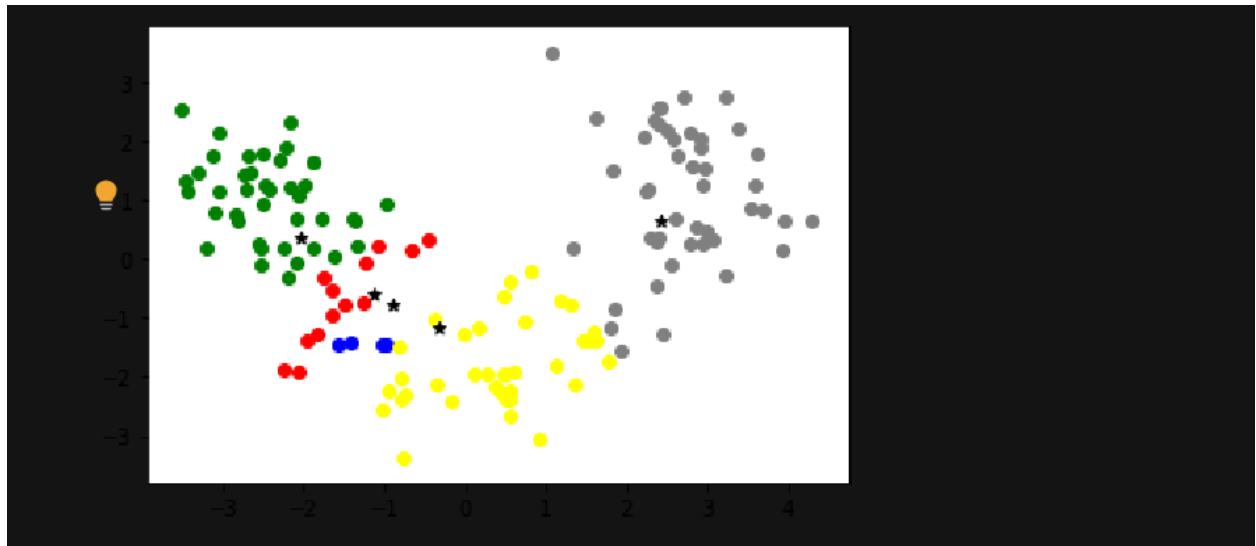
```
for kx in range(k):

    pts = np.array(clusters[kx]['points'])

    try:
        plt.scatter(pts[:,0], pts[:,1], color=clusters[kx]['color'])
    except:
        pass

    # plot the cluster center
    uk = clusters[kx]['center']
    plt.scatter(uk[0], uk[1], color='black', marker='*')

assignPointToClusters(clusters)
plotClusters(clusters)
```

d) Updated clusters after running K-means

```
In [14]: count = 1
while True:
    assignPointToClusters(clusters)
    updateClusters(clusters)
    count += 1
    if count > 1000:
        break
assignPointToClusters(clusters)
plotClusters(clusters)
updateClusters(clusters)
```

