Shirin Mohebbi **razie masoudi**
Hw #2
pattern recognition
Bayesian Classification

For this classification we use Bayesian probability:

$$P(y|\mathbf{X}) = \frac{\overbrace{P(\mathbf{X}|y)}^{Likelihood}\overbrace{P(y)}^{Prior}}{\underbrace{P(\mathbf{X})}_{normalizing\ \ factor}}$$

$$y^{new} = argmax_y\ P(y|\mathbf{X})$$

p(X) is the same for all Y so, it doesn't have any effect.
we should find p(xly) and p(y)

- Model $P(y)$ and $P(\mathbf{X}|y)$ for each class y:

$$P(y) = \phi_1^{1\{y=1\}}\phi_2^{1\{y=2\}}\cdots\phi_c^{1\{y=c\}}$$

$$P(\mathbf{X}|y=i) = \frac{1}{(\sqrt{2\pi})^n|\Sigma|^{\frac{1}{2}}}\ exp(\frac{-1}{2}(\mathbf{X}-\boldsymbol{\mu}_i)^T\Sigma^{-1}(\mathbf{X}-\boldsymbol{\mu}_i))$$

- Parameter set: $\boldsymbol{\theta} = \{\phi_1, \phi_2, ..., \phi_c, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_c, \Sigma\}$

for finding parameter i use below formula:

$$\phi_i^{MLE} = \frac{\sum_{j=1}^{m} 1\{y^{(j)} = i\}}{m}$$

$$\mu_i^{MLE} = \frac{\sum_{j=1}^{m} 1\{y^{(j)} = i\}\mathbf{X}^{(j)}}{\sum_{j=1}^{m} 1\{y^{(j)} = i\}}$$

$$\Sigma^{MLE} = \frac{1}{m}\sum_{j=1}^{m}(\mathbf{X}^{(j)} - \mu_{y^{(j)}})(\mathbf{X}^{(j)} - \mu_{y^{(j)}})^T$$

for mean:

```
self.meanClass0 = np.array( [element / self.countClass0 for element in self.sumClass0] )
self.meanClass1 = np.array( [element / self.countClass1 for element in self.sumClass1] )
```

for finding phi:

```
self.phiClass0 = self.countClass0 / self.countData
self.phiClass1 = self.countClass1 / self.countData
```

and for covariance:

```
demeanX = []
for i in range(self.countData):

  temp = []
  if self.Y[i][0] == 1:
    temp.append(self.X[i][0] - self.meanClass1[0])
    temp.append(self.X[i][1] - self.meanClass1[1])

  elif self.Y[i][0] == 0:
    temp.append(self.X[i][0] - self.meanClass0[0])
    temp.append(self.X[i][1] - self.meanClass0[1])

  demeanX.append(temp)

demeanX = np.array(demeanX)
demeanX = demeanX/np.linalg.norm(demeanX, ord=2, axis=1, keepdims=True)

self.cov = np.round( (1/self.countData) * (demeanX.T @ demeanX) , 5)
```

after i found parameters i needed, i classify X base on this formula

$$y^{new} = argmax_y \; P(y|\mathbf{X})$$

```
pXY0 = coefficient * (np.exp(  -0.5 * ( (demean0.T @ covInvers) @ demean0 )  ))
pXY1 = coefficient * (np.exp(  -0.5 * ( (demean1.T @ covInvers) @ demean1 )  ))

l0 = pXY0 * self.phiClass0
l1 = pXY1 * self.phiClass1

if (l0 > l1):
  estimate.append(0)
  if dataY[i][0] == 0:
    correct += 1
else:
  estimate.append(1)
  if dataY[i][0] == 1:
    correct += 1
```

Decision boundary :

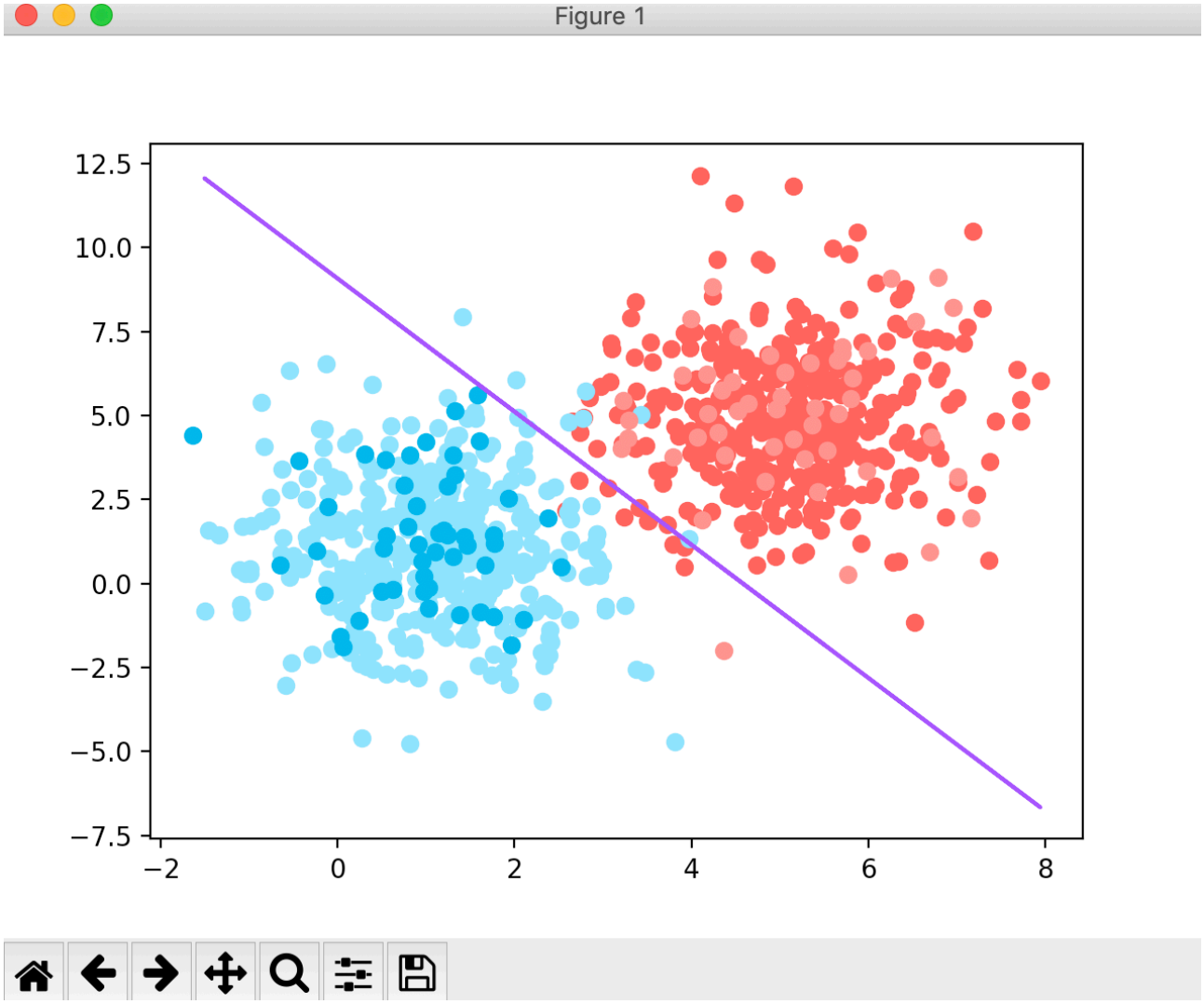
$$\Rightarrow \quad \underbrace{\mathbf{X}^T\Sigma^{-1}(\mu_i - \mu_j)}_{a\mathbf{X}} + \underbrace{\frac{1}{2}\mu_i^T\Sigma^{-1}\mu_i - \frac{1}{2}\mu_j^T\Sigma^{-1}\mu_j + log\,\frac{P(y=i)}{P(y=j)}}_{b} = 0$$

a[0] * x[0] + a[1] * x[1] + b = 0 => x1 = -(b + x0 * a[0]) / a[1]

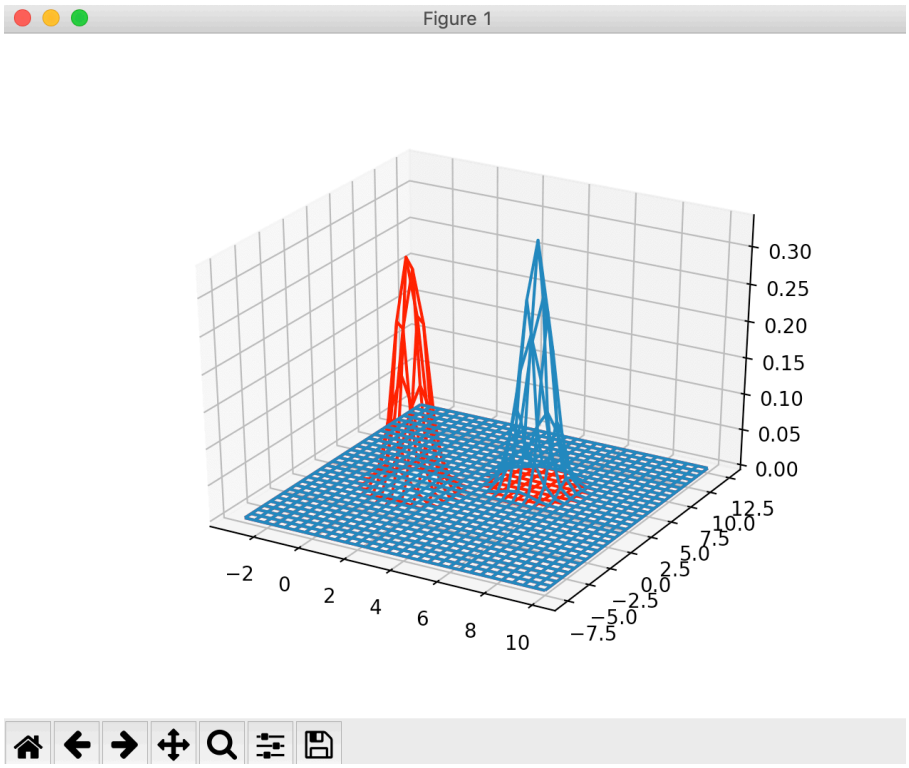
```
self.a = (np.linalg.inv(self.cov) @ (self.meanClass1 - self.meanClass0))
self.b = (0.5 * (self.meanClass0.T @ np.linalg.inv(self.cov)) @ self.meanClass0)
- (0.5 * (self.meanClass1.T @ np.linalg.inv(self.cov)) @ self.meanClass1)
+ np.log(self.phiClass0/self.phiClass1)

#a[0] * x[0] + a[1] * x[1] + b = 0    => x1 = -(b + x0 * a[0]) / a[1]
x0 = np.array([row[0] for row in self.npX])
x1 = -(self.b + x0 * self.a[0]) / self.a[1]
plt.plot(x0, x1, color = "#9933ff")
```

now lets see results for each dataset

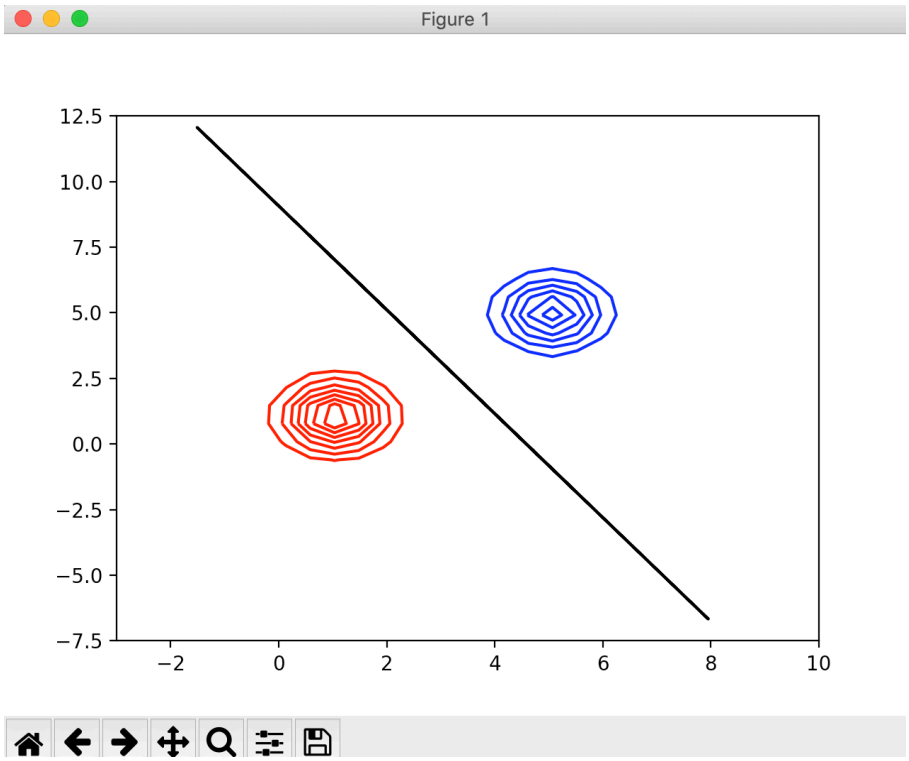
dataset1:
red dot: class1 train
light red dot: class1 test
blue dot: class2 train
light blue dot: class2 test
purple line: decision boundary



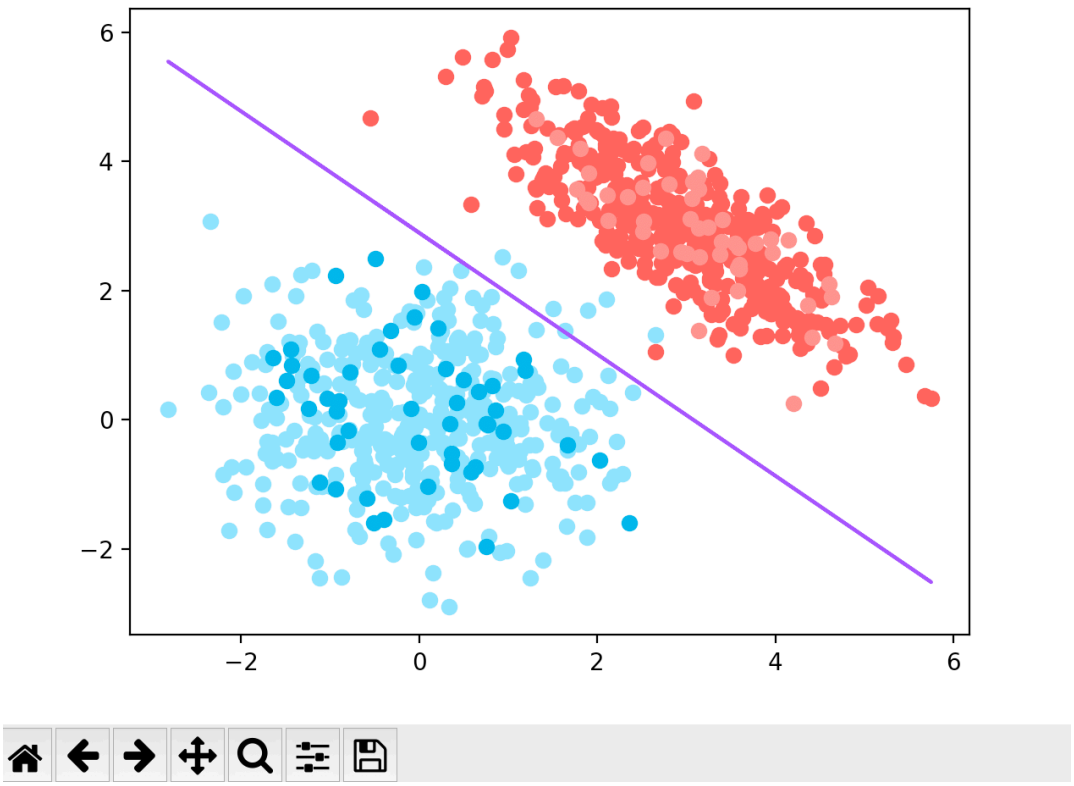
estimated PDFs



Contour estimated PDFs along with the decision boundary



dataset2


6
4
2
0
−2
−2
0
2
4
6

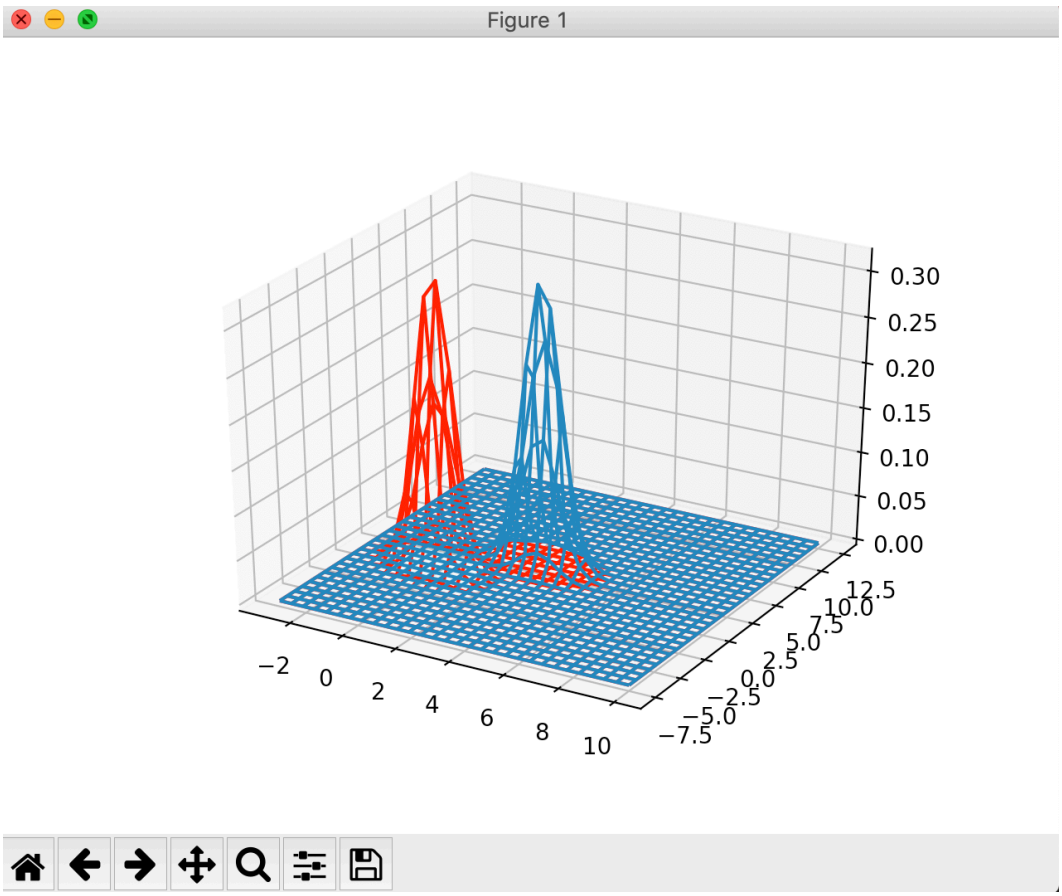
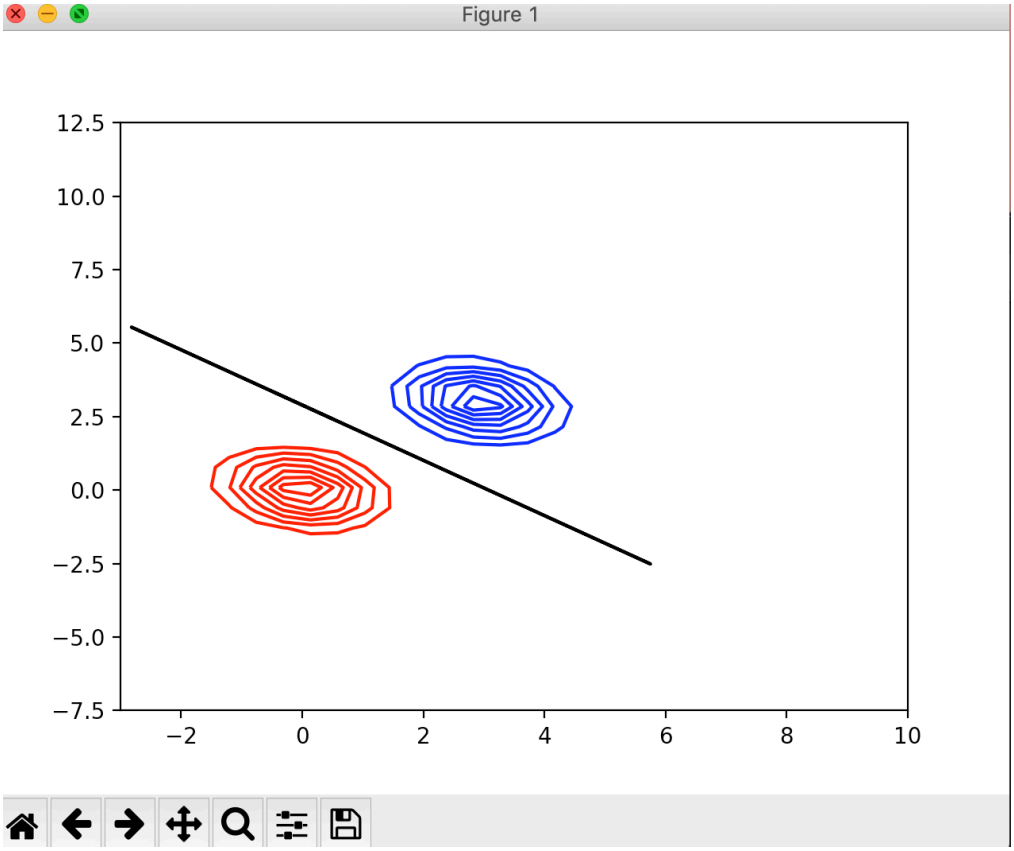


Figure 1
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−2
0
2
4
6
8
10
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5




```
train1 accuracy:
97.875 %
test1 accuracy:
99.0 %
train2 accuracy:
99.125 %
test2 accuracy:
100.0 %
```

both dataset has good accuracy
dataset1 cross section is more Circular, and dataset2 cross section is more Oval.

for implementing one vs all

we should do three model

class 1 versus class 2 and 3

class1 gets label 0

class 2 gets label 1

class 3 gets label 1

class 2 versus class 1 and 3

class1 gets label 1

class 2 gets label 0

class 3 gets label 1

class 3 versus class 1 and 2

class1 gets label 1

class 2 gets label 1

class 3 gets label 0

```
s0 = [[0]] * 40
s1 = [[1]] * 40
y1all = np.array(s0 + s1 + s1)
y2all = np.array(s1 + s0 + s1)
y3all = np.array(s1 + s1 + s0)
```

execute logistic regression for all these model and find their parameter as:

theta1vsAll

theta2vsAll

theta3vsAll

```
self.teta1all, cost1all, iteration1all = self.gradient_ascent(self.xTrain[:], y1all, initialTeta, rate, max
self.teta2all, cost2all, iteration2all = self.gradient_ascent(self.xTrain[:], y2all, initialTeta, rate, max
self.teta3all, cost3all, iteration3all = self.gradient_ascent(self.xTrain[:], y3all, initialTeta, rate, max
```

for each x, i find probability of x being in these three models. the model that has most probability of x being in its zero class is label for x.

```
def probability(self, x):
    prob = []
    prob.append(x @ self.teta1all)
    prob.append(x @ self.teta2all)
    prob.append(x @ self.teta3all)
    # print(prob)
    return prob.index(min(prob)) + 1
```

for implementing one vs one

we should do three model

class 1 versus class 2

class1 gets label 0

class 2 gets label 1

class 1 versus class 3

class1 gets label 0

class 3 gets label 1

class 2 versus class 3

class 2 gets label 0

class 3 gets label 1

```
y = np.concatenate(([[0]] * 40 , [[1]] * 40), axis=0)
x12 = self.xTrain[:80]
x13 = np.concatenate(( self.xTrain[:40], self.xTrain[80:]), axis=0)
x23 = np.concatenate(( self.xTrain[40:80], self.xTrain[80:]), axis=0)
```

execute logistic regression for all these model and find their parameter as:

theta1vs2

theta1vs3

theta2vs3

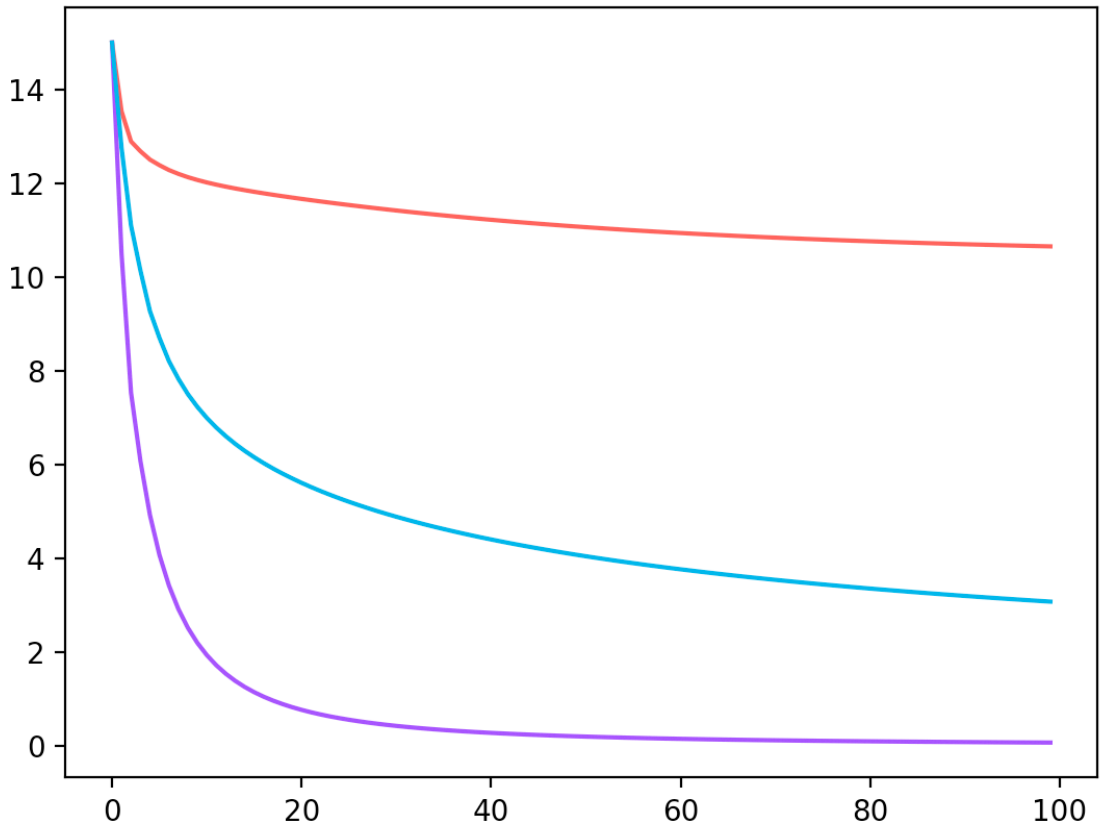
```
self.teta12, _, _ = self.gradient_ascent(x12, y[:], initialTeta, rate, maxIteration, epsilon)
self.teta13, _, _ = self.gradient_ascent(x13, y[:], initialTeta, rate, maxIteration, epsilon)
self.teta23, _, _ = self.gradient_ascent(x23, y[:], initialTeta, rate, maxIteration, epsilon)
```

for each x, each model classify x being in class0 or 1.

and between my 3 result, i choose the one with more Repetition(mode)

```
def vote(self, x):
  vote = []
  if (x @ self.teta12) < 0:
    vote.append(1)
  else:
    vote.append(2)

  if (x @ self.teta13) < 0:
    vote.append(1)
  else:
    vote.append(3)

  if (x @ self.teta23) < 0:
    vote.append(2)
  else:
    vote.append(3)

  return max(set(vote), key=vote.count)
```

cost function for enough iteration for one-vs.-all



روش one vs all: در این روش سه مدل داریم که در هر مدل یک کلاس را مقدار یک می دهیم و بقیه ی کلاس ها را مقدار صفر می دهیم سپس در هر مدل همچون مسئله ی دو کلاسه برخورد می کنیم. در مدل اول کلاس Iris-setosa را یک می دهیم و بقیه را صفر؛ سپس مقادیر سمپل ها را به صورت ستون به ستون نرمال می کنیم(feature ها مستقل از هم هستند) و ماتریس ones را برای قسمت بایاس به ماتریس X که ماتریس سمپل ها است اضافه می کنیم سپس 20 ٪ از سمپل ها را در ماتریس Xtest1 (عدد یک در Xtest نشان دهنده ی مدل یک است و به همین ترتیب در مدل های دگر) می ریزیم. سپس توابع زیر را کد می زنیم.

تابع sigmoid: این تابع مقدار predict را طبق فرمول بر می گرداند.

$$h_\theta(\mathbf{X}) = g(\theta^T\mathbf{X}) = \frac{1}{1 + e^{-\theta^T\mathbf{X}}}$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

تابع error: مقدار کل خطا را برای تک تک ردیف های ماتریس predict (مثلا در قسمت تست ماتریسY_test) و ماتریس جواب (مثلا در قسمت تست ماتریس Ytest) طبق فرمول محاسبه می کند.

تابع gradientascent: این تابع برای آپدیت مقادیر تتا است که در ابتدا تتا را ماتریسی از اعداد تصادفی بین صفر و یک می گیریم سپس طبق فرمول تتا را در هر iteration آپدیت می کنیم در این تابع همچنین مقدار خطا در هر iteration بررسی می شود، خروجی این تابع مقدار تتای آپدیت شده توسط Xtrain1 و Ytrain1 و لیستی از خطا ها در هر iteration است.

سپس مقدار تتا (تحت عنوان ماتریس new_W) را توسط Xtrain1 و Ytarin1 جهت استفاده در قسمت تست بدست می آوریم در مرحله ی بعد مدلی توسط Xtest1 و newW و تابع sigmoid تحت عنوان Y_test1 پیش بینی می کنیم سپس توسط دستور accuracy_score از کتابخانه sklearn.metrics دقت مدل را که 100 است بدست می

آوریم سپس جهت رسم نمودار cost باید لیست خطا ها در قسمت تست (error_list_test1) را با فراخوانی تابع gradientascent بدست آوریم.

تمام مراحل بالا را برای مدل های دوم و سوم طی می کنیم و نمودار Cost و دقت را در هر مدل بدست می آوریم.

سپس در انتها میانگین تمام لیست های خطا را بدست می آوریم و نمودار آن را رسم می کنیم.

پ.ن : فایل html. کد ها هم در فایل گزارش کار قرار دارد.

سافت مکس:

در این روش پیاده سازی به هر کدام از کلاس ها یک عدد اختصاص می دهیم در این مثال سه کلاس داریم که به ترتیب اعداد صفر و یک و دو را به آن ها می دهیم سپس توسط دستور get_dummies در کتابخانه pandas ماتریس one hot کلاس ها را بدست می آوریم. در مرحله ی بعد داده ها را نرمال کرده و ماتریس ones را برای قسمت بایاس به ماتریس X که ماتریس سمپل ها است اضافه می کنیم. سپس 20٪ از داده ها را به عنوان داده های تست جدا می کنیم. سپس توابع را کد می زنیم.

تابع سافت مکس: مقدار پیش بینی شده توسط مدل را طبق فرمول بر می گرداند.

تابع error : مقدار کل خطا را به این صورت محاسبه می کند که مقدار ماکزیمم در هر ردیف از ماتریس پیش بینی شده توسط مدل (Y_test) و مقدار ماکزیمم را در همان ردیف از ماتریس one hot در فرمول cost function جایگذاری می کند و مقدار کل خطا را می یابد.

O=Y_(test or train)

T=Y(test or train)

$$J(\mathbf{W}) = \frac{1}{n}\sum_{i=0}^{n} H(T_i, O_i)$$

$$H(T_i, O_i) = -\sum_{m} T_i \cdot log(O_i).$$

تابع gradientdescent: این تابع مقدار تتا (W) را آپدیت می کند بدین صورت که در ابتدا یک ماتریس تتا با مقادیر تصادفی می سازد و ماتریس ones را برای قسمت بایاس به آن اضافه می کند. سپس در یک تعداد iteration مشخص شده مقدار تتا را آپدیت می کند و مقادیر خطا را توسط فراخوانی تابع error به لیست error_list در هر iteration اضافه می کند. خروجی این تابع error_list و تتای آپدیت شده توسط ماتریس Ytrain (همان one hot

بدست آمده برای قسمت train) و ماتریس Xtrain است که این تتای خروجی را در قسمت تست در محاسبات جایگذاری می کنیم.

$$\mathbf{w}_j := \mathbf{w}_j - \alpha \nabla \mathbf{w}_j J(\mathbf{W})$$

$$j \in \{0, 1, \ldots, k\}.$$

$$\nabla \mathbf{w}_j J(\mathbf{W}) = -\frac{1}{n} \sum_{i=0}^{n} \left[\mathbf{x}^{(i)} \left(T_i - O_i \right) \right]$$

در انتها توسط تتای آپدیت شده تحت عنوان new_W و تابع سافت مکس ماتریسی تحت عنوان Y_test که همان مقادیر پیش بینی شده ما است را بدست می آوریم برای برای error_list قسمت تست تابع gradientdescent را فراخوانی می کنیم سپس نمودار آن را می کشیم و در آخر دقت مدل را این گونه محاسبه می کنیم:

در قسمت تست اندیس مقدار ماکزیمم ماتریس one hot تحت عنوان (Ytest) را با اندیس مقدار ماکزیمم ماتریس پیش بینی شده (Y_test) مقایسه می کنیم در صورتی که برابر باشند یک مقدار به correct اضافه می کنیم تا تعداد آن هایی که درست حدس زدیم را بدست آوریم سپس این مقدار را تقسیم بر تعداد کل کرده و accuracy مدل که 93.3 را بدست می آوریم.